2014

Space-efficient data structures for string searching and retrieval

Sharma Valliyil Thankachan
Louisiana State University and Agricultural and Mechanical College, sharma.thankachan@gmail.com

SPACE-EFFICIENT DATA STRUCTURES FOR STRING SEARCHING AND RETRIEVAL

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Sharma Valliyil Thankachan
B.Tech., National Institute of Technology Calicut, 2006
May 2014

Dedicated to the memory of my Grandfather.

Acknowledgments

This thesis would not have been possible without the time and effort of few people, who believed in me, instilled in me the courage to move forward and lent a supportive shoulder in times of uncertainty. I would like to express here how much it all meant to me. First and foremost, I would like to extend my sincerest gratitude to my advisor Dr. Rahul Shah. I would have hardly imagined that a fortuitous encounter in an online puzzle forum with Rahul, would set the stage for a career in theoretical computer science. His encouragement and an opportunity to work with scholarship was a dream come true. All those memories of arguing and encountering problems on algorithms, coming up with innovative results and celebrating it over drinks are moments to cherish forever. Of utmost importance is the contribution by Dr. W.-K. Hon, both for his collaboration and advise whenever I needed it. His constant encouragement and patience in proofreading my papers and correcting any errors especially at the brink of deadlines is duly appreciated. I would also like to acknowledge Dr. Jeff Vitter with whom I have collaborated on numerous papers. His guidance and knowledge has always been a source of inspiration and I feel lucky to work with him.

I would like to thank Dr. Sukhamay Kundu, Dr. Rajgopal Kannan, and Dr. Costas Busch for their willingness to be in my thesis commitee and providing valuable feedback. A special token of appreciation to my friend (and fellow PhD student) Mr. Manish Patil for guiding me with coursework and collaborating on many publications. I would also like to extend my thanks to the following professors: Dr. Stephane Durocher, Dr. Gonzalo Navarro, Dr. Moshe Lewenstein, Dr. Venkatesh Raman and Dr. Ian Munro. We have worked together on some interesting problems and I have learnt a lot from them.

Finally, I would like to express gratitude to my parents, K. K. Thankachan and Valsala Thankachan, and my sister Aswathy Thankachan for their continued emotional support. I also thank my wife Ambujam Krishnan, for believing in me at all times, and never making me feel away from home. I pray that the Almighty God will bless every one of them beyond their imaginations.

Table of Contents

List of Tables

List of Figures

Abstract

Let $\mathcal{D} = \{d_1, d_2, ..., d_D\}$ be a collection of D string documents of n characters in total, which are drawn from an alphabet set $\Sigma = [\sigma] = \{1, 2, 3, ...\sigma\}$. The *top-$k$ document retrieval* problem is to maintain $\mathcal{D}$ as a data structure, such that when ever a query $Q = (P, k)$ comes, we can report (the identifiers of) those k documents that are most relevant to the pattern P (of p characters). The relevance of a document d_r with respect to a pattern P is captured by $score(P, d_r)$, which can be any function of the set of locations where P occurs in d_r. Finding the most relevant documents to the user query is the central task of any web-search engine. In the case of web-data, the documents can be demarcated along word boundaries. All the search engines use inverted index as the back-bone data structure. For each word occurring in the document collection, the inverted index stores the list of documents where it appears. It is often augmented with relevance score and/or positional information. However, when data consists of strings (e.g., in bioinformatics or Asian language texts), there are no word demarcation boundaries and the queries are arbitrary substrings instead of being proper valid words. In this case, string data structures have to be used and central approach is to use suffix tree (or string B-tree) with appropriate augmenting data structures. The work by Hon, Shah and Vitter [34], and Navarro and Nekrich [48] resulted in a linear space data structure with optimal $O(p + k)$ query time solution for this problem. This was based on geometric interpretation of the query.

We extend this central problem, in two important areas of massive data sets. First, we consider an external memory disk based index, where we give near optimal results. Next, we consider compression aspects of data structure, reducing the storage space. This is central goal of the active research field of succinct data structures. We present several results, which improve upon several previous results, and are currently the best known space-time trade-offs in this area.

Chapter 1
Introduction

The basic task of search engines is to preprocess a text collection (referred as documents) and maintain as a data structure, so that, whenever a pattern comes as a query, the documents that are most "relevant" to the pattern (for some definition of relevance) can be identified efficiently. In Information Retrieval (IR), the most fundamental and widely used data structure for this task is the inverted index. For each word occurring in the document collection, the inverted index stores the list of documents where it appears. It is often augmented with relevance score and/or positional information. However, it applies to text collections that can be segmented into "words (terms)", so that only terms can be queried. This excludes East Asian languages such as Chinese and Korean, where automatic segmenting is an open problem, and is troublesome even in languages such as German and Finnish. A simple solution for those cases is to treat the text as a plain sequence of symbols and look for any substring in those sequences. This string model is also appealing in applications like bioinformatics and software repositories. Building a search engine over those general string collections has proved much more challenging.

The most basic string searching problem is to find all the occurrences of a pattern $P[1..p]$ in a (longer) text $\mathsf{T}[1..n]$. Earlier work has focused on developing linear-time algorithms for this problem [37]. In a data structural sense, the text is known in advance and the pattern queries arrive in an online fashion. The suffix trees [43, 66] and suffix arrays [41] are the most popular data structures to handle such queries. Both of these data structures occupy linear space (i.e., $O(n)$-word) and answer pattern matching queries in optimal $O(p + \mathtt{occ})$ time and $O(p + \log n + \mathtt{occ})$ time respectively, where $\mathtt{occ}$ is the number of occurrences of P in the text T.

Most string databases consist of a collection of strings (or documents) rather than just one single string. We shall use $\mathcal{D} = \{d_1, d_2, d_3, .., d_D\}$ for denoting the string collection of D strings of n characters in total [1]. In this case, a natural problem is to preprocess $\mathcal{D}$ and maintain it as a data structure, so that, whenever a pattern $P[1..p]$ comes as a query, those documents where P occurs at least once can be reported efficiently, instead of reporting all the occurrences. This is known as the *document listing* problem. Let $\mathtt{occ}$ be the

[1] We fix the last character of each document as \$, a special symbol that does not appear elsewhere.

number of occurrences of P over the entire collection $\mathcal{D}$, and ndoc be the number of documents where P occurs as a substring. One of the main issues is the fact that ndoc can be much smaller than occ. A simple suffix-tree-based search might be inefficient since it might involve browsing through a lot more occurrences than the actual number of qualifying documents. Therefore, it is important to design data structure which does not have to go through all the occurrences or even all the documents. The query time should be related only to ndoc. This was first addressed by Matias et al. [42], where they gave a linear-space data structure with query time $O(p \log D + \texttt{ndoc})$. Muthukrishnan [46] proposed another linear space data structure with optimal $O(p + \texttt{ndoc})$ query time.

In more realistic retrieval situations, end-users are only interested in small number (say k) of highly relevant documents from the potentially large number of documents containing the query pattern. In literature, this problem is known as top-k document retrieval, where k comes as query parameter along with the pattern P. The relevance of a document d_r with respect to a pattern P is captured using $score(P, d_r)$, which can be any function of the set of occurrences (given by their locations) of P in document d_r. For example, $score(P, d_r)$ can simply be the term frequency $\texttt{TF}(P, d_r)$ (i.e., the number of occurrences of P in d_r), or it can be the term proximity $\texttt{TP}(P, d_r)$ (i.e., the distance between the pair of closest occurrences of P in d_r), or a pattern-independent *importance* score of d_r such as PageRank [56]. Formally, we have the following definition.

Problem 1. Top-k Document Retrieval: *Let $\mathcal{D} = \{d_1, d_2, ..., d_D\}$ be a collection of D string documents of n characters in total, which are drawn from an alphabet set $\Sigma = [\sigma] = \{1, 2, 3, ...\sigma\}$. The* top-$k$ document retrieval *problem is to maintain $\mathcal{D}$ as a data structure, such that when ever a query $Q = (P, k)$ comes, we can report (the identifiers of) those k documents with the highest $score(P, d_r)$ values. Here, $score(P, d_r)$ is any function which is dependent only on the set of occurrences of P in d_r.*

If we are required to report the answers in the decreasing order of $score(P, \cdot)$, we call it as *sorted retrieval*, otherwise it is called *unsorted retrieval*. Throughout the rest of this thesis, unless otherwise specified, we assume sorted retrieval. Also, the problem is known as top-k frequent document retrieval, if $score(\cdot, \cdot) = \texttt{TF}(\cdot, \cdot)$. Hon, Shah and Vitter [34] gave the first framework to answer such a retrieval query based on augmented suffix trees. Their work along with subsequent improvement by Navarro and Nekrich [48] resulted in $O(n)$-word (linear) space data structure and optimal $O(p + k)$ query time solution for

this problem. These results are based on a reduction of *top-k document retrieval* problem to a geometric problem known as 4-sided range reporting in 3d (three dimensions). Although, the general case of this geometric problem is hard, results in [34] are based on a crucial observation that the problem in hand can be decomposed into $O(p)$ subproblems, each of them are 3-sided range reporting queries in 2d (which can be solved optimally). This inevitably adds an additive $O(p)$ term in the query time, however it will not change the query time as $O(p)$ time is needed for initial pattern search using suffix tree.

Historically, both suffix tree and suffix array data structures are considered to consume "linear" space. However, the notion of space measure here was in terms of memory words. When measured in terms of bits, these data structures take $O(n \log n)$ bits, which is asymptotically higher than the $n \log \sigma$ bits required to store the text in plain form; here, $\Sigma = [\sigma] = \{1, 2, ..., \sigma\}$ denotes the alphabet set from which characters of T and P are drawn. Practically for DNA texts (with $\Sigma = \{A, G, C, T\}$), the suffix tree (resp., suffix array) structure is reported to take 15 to 50 (resp., 4 to 15) times more space than the original data. For example, consider human genome data (which takes roughly 3-4 GB space) can be accommodated in main memory of the typical modern computer systems. However, a suffix tree or suffix array data structure built over it may not. Furthermore, the text can often be compressed by entropy-compression methods like gzip or bzip. Thus, the actual gap between the indexing space and the storage space is even larger.

A longstanding open question was to develop data structures for string searching applications, which takes space close to the compressed representation of string itself. This was answered positively by Grossi and Vitter [27] using their Compressed Suffix Array (CSA) data structure, and Ferragina and Manzini [17] by their data structure called FM-index; subsequently, an exciting field of compressed text indexing was established. Many versions of CSA and FM-index were later proposed. A space optimal FM-index by Ferragina et al. [18] occupy $nH_h + o(n \log \sigma)$ bits, where $H_h \leq \log \sigma$ denotes the empirical hth-order entropy of T [2] and can answer queries in $O((p + occ \log^{1+\epsilon} n)(1 + \log \sigma / \log \log n))$ time (we defer more details to Section 2.3).

When it comes to the task of designing a compressed/succinct space data structure for the top-k document retrieval problem, the corresponding suffix tree data structure can be replaced by its compressed version. However, the challenging part is how to compress the augmented information. One of the technique is *sparsification*, which is based on the principle of "store less, do more while querying". The data structures are build for the case where $score(P, d_r)$ is the number of occurrences of P in d_r. The first data structure

[2]The space bound holds for all $h < \alpha \log n / \log \sigma$, where α is any fixed constant with $0 < \alpha < 1$.

of this category is proposed by Hon et al. [34], and it takes $2nH_h + o(n \log \sigma) + O(D \log n)$ bits space and $O((p + k \log^{4+\epsilon} n)(1 + \log \sigma / \log \log n))$ query time, where ϵ is any positive constant. This thesis present a result, which improves this query time to $O((p + k \log k \log^{1+\epsilon} n)(1 + \log \sigma / \log \log n))$, which is currently the fastest data structure within this space bound.

A data structure taking roughly $nH_k + n \log D + o(n \log D + n \log \sigma)$ bits of space is referred as a compact space data structure. For unsorted top-k document retrieval, we present a compact space solution of query time $O(p + k \log^* k)$ (detailed discussion on other known compact space data structures is deferred to Chapter 5).

With the advent of enterprise search, deep desktop search, email search technologies, the data structures which reside on disks are more and more important. Many biological databases are now being tuned for external memory versions as the amount of sequence data grows. Importance of disk resident data structures is highlighted by the fact that even the compressed data structures built over massive string data (for example, 1000 genome project) may not fit in internal memory. Despite these motivations as I/O-efficient data structure for string retrieval has been elusive. Here also the suffix tree data structure can be replaced by its external memory counter part called string B-tree [16]. However, the challenging part is how to query on the augmented information in optimal I/Os. The desired optimal I/O bound of $O(p/B + \log_B n + k/B)$ cannot be achieved by Hon et al.'s approach, because it inevitably adds an $O(p)$ additive term. Therefore, instead of modeling as a general 4-sided orthogonal range query, we exploit several special properties of this problem that allows us to be able to break this barrier.

1.1 The Models of Computation

The data structures presented in this thesis work are in either Word-RAM model, or in external memory (also known as I/O model) [2]. In Word-RAM model, we assume that the memory is partitioned into continuous blocks (or words). At any point of time, we can load data into a block or access data within a block in constant time. We always assume the size of a block is $\Theta(\log n)$ bits, where n denotes the size of the problem in hand.

The external-memory model or I/O model was introduced by Aggarwal and Vitter [2] in 1988. In this model, the CPU is connected directly to an internal memory of size M, which is then connected to a much larger and slower disk. The disk is divided into blocks of B words (i.e., $\Theta(B \log n)$ bits). The CPU can only operate on data inside the internal memory. So, we need to transfer data between internal memory and disk

through I/O operations, where each I/O may transfer a block from the disk to the memory (or vice versa). Since internal memory (RAM) is much faster, operations on data inside this memory are considered free. Performance of an algorithm is measured by the number of I/O operations used.

1.2 Our Contributions

In this thesis, we present several data structures for answering top-k document retrieval queries in various settings, and our main results are summarized below.

- We present an I/O-optimal external memory data structure of $O(n \log^* n)$-word space for the (unsorted) top-k document retrieval problem. Here $\log^* n$ represents the iterative logarithm of n. Our data structure can be easily adapted to internal memory and achieve $\Theta(n)$-word space and $O(p + k)$ query time.

- The lower bound on the minimum space required for maintaing $\mathcal{D}$ is $nH_k + D \log(n/D) + O(D)$ bits. For the case where $score(P, d_r)$ is the number of occurrences of P in d_r, we present a data structure occupying $2nH_k + D \log(n/D) + O(D)$ bits of space, and can answer queries in $O((p + k \log k \log^{1+\epsilon} n)(1 + \log \sigma / \log \log n))$ time, where ϵ is any positive constant. Notice that the earlier data structure occupying the same space required $O((p + k \log^{4+\epsilon} n)(1 + \log \sigma / \log \log n))$ time.

- For the case where $score(P, d_r)$ is the number of occurrences of P in d_r, we present a compact space data structure of $nH_k + n \log D + o(n \log D + n \log \sigma)$ bits space and $O(p + k \log^* k)$ query time for (unsorted) top-k document retrieval queries. That is $O(\log^* k)$ per document retrieval time. Notice that the earlier best known data structure required $O((\log \sigma \log \log n)^{1+\epsilon})$ time per reported document.

- We provide a framework to answer top-k queries for two or more patterns. For two patterns P_1 and P_2 of lengths p_1 and p_2 respectively, we derive linear-space (i.e., using $O(n)$ words) indexes with query time $O(p_1 + p_2 + \sqrt{nk \log n} \log \log n)$ for various score functions.

1.3 Roadmap

Chapter 2 gives the preliminaries. Next we describe our external memory result in Chapter 3. Chapter 4 and Chapter 5 are dedicated to the description of our results on succinct space and compact space data structures. In Chapter 6, we introduce the several problems and the corresponding solutions for an extension of top-k retrieval, where a query may consists of more than one pattern. Finally, we conclude in Chapter 7 with some open problems.

Chapter 2
Preliminaries

In this section, we briefly describe various known data structures that are the building blocks of our newly introduced data structures. Also, the notations and definitions introduced in this section will be followed throughout the rest of this thesis.

2.1 Generalized Suffix Tree

Let $\mathsf{T} = d_1 d_2 d_3 \cdots d_D$ be the text (of n characters from an alphabet set $\Sigma = [\sigma]$) obtained by concatenating all the documents in $\mathcal{D}$. The last character of each document is \$, a special symbol that does not appear anywhere else in T. Each substring $\mathsf{T}[i..n]$, with $i \in [1, n]$, is called a *suffix* of T. The *generalized suffix tree* (GST) of $\mathcal{D}$ is a lexicographic arrangement of all these n suffixes in a compact trie structure, where the ith leftmost leaf represents the ith lexicographically smallest suffix. Each edge in GST is labeled by a string, and $path(x)$ of a node x is the concatenation of edge labels along the path from the $root$ of GST to node x. Let ℓ_i for $i \in [1, n]$ represent the ith leftmost leaf in GST. Then $path(\ell_i)$ represents the ith lexicographically smallest suffix of T. Corresponding to each node, a perfect hash function [20] is maintained such that, given any node u and any character $c \in \Sigma$, we can compute the child node v of u (if it exists) where the first character on the edge connecting u and v is c. A node x is called the *locus* of a pattern P, if it is the highest node $path(x)$ prefixed by P. The total space consumption of GST is $O(n)$ words and the time for computing the locus node of P is $O(p)$. When $\mathcal{D}$ contains only one document d_r, the corresponding GST is commonly known as the *suffix tree* of d_r [66].

2.2 Suffix Array

The *suffix array* $\mathsf{SA}[1..n]$ is an array of length n, where $\mathsf{SA}[i]$ is the starting position (in T) of the ith lexicographically smallest suffix of T [41]. In essence, the suffix array contains the leaf information of GST but without the tree structure. An important property of SA is that the starting positions of all the suffixes with the same prefix are always stored in a contiguous region of SA. Based on this property, we define the *suffix range* of P in SA to be the maximal range $[sp, ep]$ such that for all $i \in [sp, ep]$, $\mathsf{SA}[i]$ is the starting point of a suffix of T prefixed by P. Therefore, ℓ_{sp} and ℓ_{ep} represents the first and last leaves in the subtree of the locus node of P in GST.

2.3 Compressed Suffix Arrays

A compressed representation of suffix array is called a *compressed suffix array* (CSA) [27, 17, 26]. We denote the size (in bits) of a CSA by $|\mathsf{CSA}|$, the time for computing $\mathsf{SA}[\cdot]$ and $\mathsf{SA}^{-1}[\cdot]$ values by t_{sa}, and the time for finding the suffix range of a pattern of length p by $t_s(p)$. There are various versions of CSA in the literature that provide different performance tradeoffs (see [47] for an excellent survey). For example, the space-optimal CSA by Ferragina et al. [18] takes $nH_h + o(n \log \sigma)$ bits space, where $H_h \leq \log \sigma$ denotes the empirical hth-order entropy of T.[1] The timings t_{sa} and $t_s(p)$ are $O(\log^{1+\epsilon} n \log \sigma)$ and $O(p(1 + \log \sigma / \log \log n))$, respectively. Recently, Belazzougui and Navarro [4] proposed another CSA of space $nH_h + O(n) + o(n \log \sigma)$ bits with $t_s(p) = O(p)$ and $t_{sa} = O(\log n)$.

2.4 Bit Vectors with Rank/Select Support

Let $\mathsf{B}[1..n]$ be a bit vector with its m bits set to 1. Then, $rank_{\mathsf{B}}(i)$ represents the number of 1's in $\mathsf{B}[1..i]$ and $select_{\mathsf{B}}(j)$ represents the position in B where the jth 1 occurs (if $j > m$, return NIL). The minimum space needed for representing B is given by $\lceil \log \binom{n}{m} \rceil \leq m \log(ne/m) = m \log(n/m) + 1.44m$ [57]. There exists representations of B in $n + o(n)$ bits and $m \log(n/m) + O(m) + o(n)$ bits, which can support both $rank_{\mathsf{B}}(\cdot)$ and $select_{\mathsf{B}}(\cdot)$ operations in constant time. These structures are known as fully indexible dictionaries. Another representation, where the space occupancy is $m \log(n/m) + O(m)$ bit support only $select_{\mathsf{B}}(\cdot)$ operation in constant time is known as indexible dictionary [61].

2.5 Succinct Representation of Ordinal Trees

The lower bound on the space needed for representing any n-node ordered rooted tree, where each node is labeled by its preorder rank in the tree, is $2n - O(\log n)$ bits. Using succinct data structure occupying $o(n)$ bits extra space, the following operations can be supported in constant time [63]: (i) $parent(u)$, which returns the parent of node u, (ii) $lca(u, v)$, which returns the lowest common ancestor of two nodes u and v, and (iii) $lmost\_leaf(u)/rmost\_leaf(u)$, which returns the leftmost/rightmost leaf of node u.

2.6 Document Array

The document array $E\ [1..n]$ is defined as $E\ [j] = r$ if the suffix $\mathsf{T}[\mathsf{SA}[j]..n]$ belongs to document d_r. Moreover, the corresponding leaf node ℓ_j is said to be *marked* with document d_r.

[1] The space bound holds for all $h < \alpha \log n / \log \sigma$, where α is any fixed constant with $0 < \alpha < 1$.

By maintaining E using the structure described in [25], we have the following result.

Lemma 1. The document array E can be stored in $n \log D + o(n \log D)$ bits and support $rank_E$, $select_E$ and $access_E$ operations in $O(\log \log D)$ time, where

- $rank_E\ (r, i)$ returns the number of occurrences of r in $E\ [1..i]$;
- $select_E\ (r, j)$ returns the location of jth leftmost occurrence of r in E ; and
- $access_E\ (i)$ returns $E[i]$.

Define a bit-vector $\mathsf{B_E}[1..n]$ such that $\mathsf{B_E}[i] = 1$ if and only if $\mathsf{T}[i] = \$$. Then, the suffix $\mathsf{T}[i..n]$ belongs to document d_r if $r = 1 + rank_{\mathsf{B_E}}(i)$, where $rank_{\mathsf{B_E}}(i)$ represents the number of 1s in $\mathsf{B_E}[1..i]$. The following is another useful result.

Lemma 2. Using CSA and an additional structure of size $|\mathsf{CSA}^*| + D \log \frac{n}{D} + O(D) + o(n)$ bits, the document array E can be simulated to support $rank_E$ operation in $O(t_{sa} \log \log n)$ time, and $select_E$ and $access_E$ operations in $O(t_{sa})$ time.

Proof. The document array E can be simulated using the following structures: **(i)** compressed suffix array CSA of T (of size $|\mathsf{CSA}|$ bits), where $\mathsf{SA}[\cdot]$ and $\mathsf{SA}^{-1}[\cdot]$ represent the suffix array and inverse suffix array values in CSA; **(ii)** compressed suffix array CSA_r of document d_r (of size $|\mathsf{CSA}_r|$ bits) corresponding to every $d_r \in \mathcal{D}$, where $\mathsf{SA}_r[\cdot]$ and $\mathsf{SA}_r^{-1}[\cdot]$ represent the suffix array and inverse suffix array values in CSA_r; and **(iii)** the bit-vector $\mathsf{B_E}$ maintained in $D \log \frac{n}{D} + O(D) + o(n)$ bits with constant-time rank/select supported [61]. Hence the total space is bounded by $|\mathsf{CSA}^*| + D \log \frac{n}{D} + O(D) + o(n)$ bits in addition to the $|\mathsf{CSA}|$ bits of CSA, where $|\mathsf{CSA}^*| = \max\{|\mathsf{CSA}|, \sum_{r=1}^{D} |\mathsf{CSA}_r|\}$ [2].

The function $access_E\ (i) = 1 + rank_{\mathsf{B_E}}(\mathsf{SA}[i])$ can be computed in $O(t_{sa})$ time. For computing $select_E\ (r, j)$, we first compute the jth smallest suffix in CSA_r and obtain the position pos of this suffix within document d_r, from which we can easily obtain the position pos' of this suffix within T as $select_{\mathsf{B_E}}(r-1) + pos$, where $select_{\mathsf{B_E}}(x)$ is the position of the xth 1 in $\mathsf{B_E}$. After that, we compute $\mathsf{SA}^{-1}[pos']$ in CSA as the desired answer for $select_E\ (r, j)$. This takes $O(t_{sa})$ time. The function $rank_E\ (r, i) = j$ can be obtained in $O(t_{sa} \log n)$ time using a binary search on j such that $select_E\ (r, j) \leq i < select_E\ (r, j+1)$. Belazzougui et al. [6] showed that the time for computing $rank_E\ (r, i)$ can be improved to $O(t_{sa} \log \log n)$ as follows: At every $(\log^2 n)$th leaf

[2]Notice that in the case of some specific versions of CSA, $|\mathsf{CSA}^*|$ can be bounded by $|\mathsf{CSA}| + O(D \log n)$.

of each CSA_r, we explicitly maintain its corresponding position in CSA and a predecessor search structure over it [67]; the size of this additional structure is $o(n)$ bits. Now, when we answer the query, we can first search this predecessor structure for an approximate answer, and the exact answer can be obtained by a binary search on a smaller range of only $\log^2 n$ leaves. □

Lemma 3. Let E be the document array corresponding to a document collection $\mathcal{D}$. Then, for any document $d_r \in \mathcal{D}$, $\text{TF}(P, d_r) = rank_E\ (r, ep) - rank_E\ (r, sp - 1)$, where $[sp, ep]$ represents the suffix range of P.

2.7 Differentially Encoding a Sorted Array

Let $\mathsf{A}[1..m]$ be an array of integers such that $1 \leq \mathsf{A}[i] \leq \mathsf{A}[i+1] \leq n$. The array A can be encoded using a bitmap $\mathsf{B} = 10^{c_1}10^{c_2}10^{c_3}\cdots 10^{c_n}$, where c_i denotes the number of entries $\mathsf{A}[\cdot] = i$. The length of B is $m + n$, and hence B can be maintained in $(m + n)(1 + o(1))$ bits (along with constant-time rank/select structures [44, 11]). Then, for any given $j \in [1, m]$, we can compute $A[j]$ in constant time by first finding the location of the jth 0 in B, and then counting the number of 1s up to that position.

2.8 String B-tree

String B-tree (SBT) [16] is a data structure for a text T that supports efficient online pattern matching queries in the external-memory setting. Basically, it is a B-tree over the suffix array SA of T but with extra information stored in each B-tree node to facilitate the matching. The performance of SBT is summarized as follows.

Lemma 4. Given a text T of length n characters, we can build a string B-tree data structure in $\Theta(n/B)$ blocks or $\Theta(n \log n)$ bits such that the suffix range of any input pattern P can be obtained in $O(p/B + \log_B n)$ I/Os, where p is the number of characters of P and B denotes the block size.

Chapter 3
External Memory Data Structures

In this chapter [1], we present our new framework for top-k document retrieval. Based on this, we derive the first non-trivial external memory data structure as described in the following theorem

Theorem 1. *In the external memory model, there exists an $O(nh)$-word date structure that can answer the (unsorted) top-k document retrieval queries in $O(p/B + \log_B n + \log^{(h)} n + k/B)$ I/Os for any $h \leq \log^* n$, where $\log^{(h)} n = \log\log^{(h-1)} n$, $\log^{(1)} n = \log n$ and B is the block size.*

Corollary 1. *There exists an $O(n \log^* n)$-word structure for answering the (unsorted) top-k document retrieval queries in optimal $O(p/B + \log_B n + k/B)$ I/Os.*

3.1 Preliminary: Top-k Framework

This section briefly explains the linear space framework for top-k document retrieval based on the work of Hon et al. [34], and Navarro and Nekrich [48]. We first build a generalized suffix tree (GST) of a document collection $\mathcal{D}= \{d_1, d_2, d_3, \ldots, d_D\}$. The definitions of $path, depth$ and $locus$ remains the same as we described in Section 2.1. The locus node of P is denoted by u_P. By numbering all the nodes in GST in the pre-order traversal manner, the part of GST relevant to P (i.e., the subtree rooted at u_P) can be represented as a range.

Nodes are marked with documents. A leaf node ℓ is marked with a document $d \in \mathcal{D}$ if the suffix represented by ℓ belongs to d. An internal node u is marked with d if it is the lowest common ancestor of two leaves marked with d. Notice that a node can be marked with multiple documents. For each node u and each of its marked documents d, define a *link* to be a quadruple $(origin, target, doc, score)$, where $origin = u$, $target$ is the lowest proper ancestor of u marked with d, $doc = d$ and $score = w\big(path(u), d\big)$. Two crucial properties of the links identified in [34] are listed below.

Lemma 5. *For each document d that contains a pattern P, there is a unique link whose origin is in the subtree of u_P and whose target is a proper ancestor of u_P. The score of the link is the score of d w.r.t. P.*

[1] This section previously appeared as, Rahul Shah, Cheng Sheng and Sharma V. Thankachan and Jeffrey Scott Vitter, *Top-k Document Retrieval in External Memory*, In Proceedings of European Symposium on Algorithms (ESA), 2013, pages 803-814. It is reprinted by permission of Springer.

Lemma 6. *The total number of links is* $O(n)$.

In fact the number of links is exactly equal to number of nodes in the suffix tree of d_i over all i's in $[1, D]$. We refer to [34] for the detailed proof of this result. Based on Lemma 5, the top-k document retrieval problem can be reduced to the problem of finding the top-k links (according to its score) stabbed by u_P, where *link stabbing* is defined as follows:

Definition 1 (Link Stabbing)**.** *We say that a link is* stabbed *by node* u *if it is originated in the subtree of* u *and targets at a proper ancestor of* u.

If we order the nodes in GST as per the *pre-order traversal* order, these constraints translate into finding all the links (i) the numbers of whose origins fall in the number range of the subtree of u_P, and (ii) the numbers of whose targets are less than the number of u_P. Regarding constraint (i) as a two-sided range constraint on x-dimension, and regarding constraint (ii) as a one-sided range constraint on y-dimension, the problem asks for the top-k weighted points that fall in a three-sided window in 2d space, where weight of a point is the score of the corresponding link [48].

The framework of Hon et al. [34] takes linear space and answers the query in $O(p+k \log k)$ time. This was then improved by Navarro and Nekrich [48] to achieve $O(p + k)$ query cost. Both [34] and [48] reduced this problem to a 4-sided orthogonal range query in 3-dimension, which is defined as follows: the data consists of a set S of 3-dimensional points and the query consists of four parameters x', x'', y' and z', and output is the set of all those points $(x_i, y_i, z_i) \in S$ such that $x_i \in [x', x'']$, $y_i \leq y'$ and $z_i \geq z'$. While general 4-sided orthogonal range searching is proved hard [9], the desired bounds can nevertheless be achieved by identifying a special property that one dimension of the reduced subproblem can only have p distinct values. Even though there has been series of work on top-k string, including in theory as well as practical IR communities, most implementations (as well as theoretical results) have focused on RAM based compressed and/or efficient data structures (See the recent surveys [50, 29]). In this Section, we present an alternative framework for solving this problem and obtain the first non-trivial external memory [2].

3.2 External Memory Structures

This section is dedicated for proving Theorem 1. The initial phase of pattern search can be performed in $O(p/B + \log_B n)$ I/O's using a string B-tree [16]. Once the suffix range of P is identified, we take the lowest common ancestor of the left-most and right-most leaves in the suffix range of GST to identify the

locus node u_P. Hence, the first phase (i.e., finding the locus node u_P of P) takes optimal I/O's and now we focus only on the second phase (i.e., reporting the top-k links stabbed by u_P). Instead of solving the top-k version, we first solve a threshold version in Sec 3.2.1 where the objective is to retrieve those links stabbed by u_P with $score$ at least a given threshold τ. Then in Sec 3.2.2, we propose a separate structure that converts the original top-k-form query into a threshold-form query so that the structure in Sec 3.2.1 can now be used to answer the original problem. Finally, we obtain Theorem 1 via bootstrapping on a special structure for handling top-k queries in lesser number of I/Os for small values of k. We shall assume all scores are distinct and are within $[1, O(n)]$. Otherwise, the ties can be broken arbitrarily and reduce the values into rank-space.

3.2.1 Breaking Down into Sub-Problems

Instead of solving the top-k version, we first solve a threshold version, where the objective is to retrieve those links stabbed by u_P with $score$ at least a given threshold τ. We show that the problem can be decomposed into simpler subproblems, which consists of a 3d dominance reporting and $O(\log(n/B))$ 3-sided range reporting in 2d, both can be solved efficiently using known structures. The main result is captured in Lemma 7 defined below. From now onwards, the origin, target and score of a link L_i are represented by o_i, t_i and w_i respectively.

Lemma 7. *There exists an $O(n)$ space data structure for answering the following query: given a query node u_P and a threshold τ, all links stabbed by u_P with score $\geq \tau$ can be reported in $O(\log^2(n/B) + z/B)$ I/Os, where z is the number of outputs.*

For any node u in GST, we use u to denote its pre-order rank as well. Let $size(u)$ denotes the number of leaves in the subtree of u, then we define its *rank* as:

$$rank(u) = \lfloor \log \lceil \frac{size(u)}{B} \rceil \rfloor$$

Note that $rank(.) \in [0, \lfloor \log \lceil \frac{n}{B} \rceil \rfloor]$. A contiguous subtree consisting of nodes with the same rank is defined as a *component*, and the $rank$ of a component is same as the rank of nodes within it. Therefore, a *component* with $rank = 0$ is a bottom level subtree of size (number of leaves) at most B. From the definition, it can be seen that a node and at most one of its children can have the same $rank$. Therefore, a component with $rank \geq 1$ consists of nodes in a path which goes top-down in the tree.

The number of links originating within the subtree of any node u is at most $2size(u) - 1$. Therefore, the number of links originating within a component with $rank = 0$ is $O(B)$. These $O(B)$ links corresponding to each component with $rank = 0$ can be maintained separately as a list, taking total $O(n)$ words space. Now, given a locus node u_P, if $rank(u_P) = 0$, the number of links originating within the subtree of u_P is also $O(B)$ and all of them can be processed in $O(1)$ I/O's by simply scanning the list of links corresponding to the component to which u_P belongs to. The query processing is more sophisticated when $rank(u_P) \geq 1$. For handling this case, we classify the links into the following 2 types based on the $rank$ of its target with respect to the $rank$ of query node u_P: ***equi-ranked links***: links with $rank(target) = rank(u_P)$ and ***high-ranked links***: links with $rank(target) > rank(u_P)$.

Next we show that the problem of retrieving outputs among equi-ranked links can be reduced to a 3d dominance query, and the problem of retrieving outputs among high-ranked links can be reduced to at most $\lfloor \log \lceil \frac{n}{B} \rceil \rfloor$ 3-sided range queries in 2d.

3.2.1.1 Processing Equi-ranked Links.

Let C be a component and S_C be set of all links L_i, such that its target t_i is a node in C. Also, for any link $L_i \in S_C$, let pseudo_origin s_i be the (pre-order rank of) lowest ancestor of its origin o_i within C (see Figure 3.1). Then a link $L_i \in S_C$ originates in the subtree of any node u within C if and only if $s_i \geq u$. Now if the locus u_P is a node in C, then among all equi-ranked links, we need to consider only those links $L_i \in S_C$, because the origin o_j of any other equi-ranked link $L_j \notin S_C$, will not be in the subtree of u_P. Based on the above observations, all equi-ranked output links are those $L_i \in S_C$ with $t_i < u_P \leq s_i$ and $w_i \geq \tau$. To solve this in external memory, we treat each link $L_i \in S_C$ as a 3d point (t_i, s_i, w_i) and maintain a 3d dominance query structure over it. Now the outputs with respect to u_P and τ are those links corresponding to the points within $(-\infty, u_P) \times [u_P, \infty) \times [\tau, \infty)$. Such a structure for S_C can be maintained in linear $O(|S_C|)$ words of space and can answer the query in $O(\log_B |S_C| + z_{eq}/B)$ I/O's using the result by Afshani [1], where $|S_C|$ is the number of points (corresponding to links in S_C) and z_{eq} be the output size. Thus overall these structures occupies $O(n)$-word space.

Lemma 8. *Given a query node u_P and a threshold τ, all the equi-ranked links stabbed by u_P with score $\geq \tau$ can be retrieved in $O(\log_B n + z_{eq}/B)$ I/Os using an $O(n)$ word space data structure, where z_{eq} is the output size.* □

3.2.1.2 Processing High-ranked Links.

The following is an important observation.

Observation 1. *Any link L_i with its origin o_i within the subtree of a node u is stabbed by u if $rank(t_i) > rank(u)$, where t_i is the target of L_i.*

This implies, while looking for the outputs among the high-ranked links, the condition of t_i being a proper ancestor of u_P can be ignored as it is taken care of automatically if $o_i \in \left[u_P, u'_P\right]$, where u'_P be the (pre-order rank of) right-most leaf in the subtree rooted at u_P. Let G_r be the set of all links with rank equals r for $1 \leq r \leq \lfloor \log \lceil \frac{n}{B} \rceil \rfloor$. Since there are only $O(\log(n/B))$ sets, we shall maintain separate structures for links in each G_r by considering only $origin$ and $score$ values. We treat each link $L_i \in G_r$ as a $2d$ point (o_i, w_i), and maintain a 3-sided range query structure over them for $r = 1, 2, .., \lfloor \log \lceil \frac{n}{B} \rceil \rfloor$. All high-ranked output links can be obtained by retrieving those links in $L_i \in G_r$ with the corresponding point $(o_i, w_i) \in [u_P, u'_P] \times [\tau, \infty]$ for $r = rank(u_P) + 1, .., \lfloor \log \lceil \frac{n}{B} \rceil \rfloor$. By using the linear space data structure in [3], the space and I/O bounds for a particular r is given by $O(|G_r|)$ words and $O(\log_B |G_r| + z_r/B)$, where z_r is the number of output links in G_r. Since a link can be a part of at most one G_r, the total space consumption is $O(n)$ words and the total query I/Os is $O(\log_B n \log(n/B) + z_{hi}/B) = O(\log^2(n/B) + z_{hi}/B)$, where z_{hi} represents the number of high-ranked output links.

Lemma 9. *Given a query node u_P and a threshold τ, all the high-ranked links stabbed by u_P with score $\geq \tau$ can be retrieved in $O(\log^2(n/B) + z_{hi}/B)$ I/Os using an $O(n)$ word space data structure, where z_{hi} is the output size.* □

By combining Lemma 8 and Lemma 9, we obtain Lemma 7.

3.2.2 Converting Top-k to Threshold via Logarithmic Sketch

Here we derive a linear space data structure, such that given a query node u and a parameter k, a threshold τ can be computed in constant I/Os, such that the number of links z stabbed by u with score $\geq \tau$ is bounded by, $k \leq z \leq 2k + O(\log n)$. Hence query I/Os in Lemma 7 can be modified as $O(\log^2(n/B) + z/B) = O(\log^2(n/B) + k/B)$. From the retrieved z outputs, the actual top-k answers can be computed by selection and filtering in another $O(z/B) = O(k/B + \log_B n)$ I/O's. We summarize our result in the following lemma.

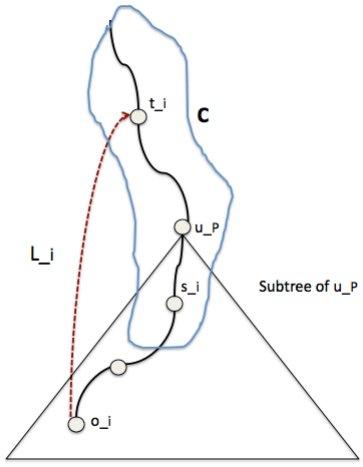

FIGURE 3.1. Pseudo Origin

Lemma 10. *There exist an $O(n)$ word data structure for answering the following query in $O(\log^2(n/B) + k/B)$ I/O's: given a query point u and an integer k, report the top-k links stabbed by u.* □

We now give the details of top-k to threshold conversion. First, identify certain nodes in the GST as marked nodes and prime nodes with respect to a parameter g called the *grouping factor*. The procedure starts by combining every g consecutive leaves (from left to right) together as a group, and marking the lowest common ancestor (LCA) of first and last leaf in each group. Further, we mark the LCA of all pairs of marked nodes recursively. Additionally, we ensure that the root is always marked. At the end of this procedure, the number of marked nodes in GST will be $O(n/g)$ [34]. Prime nodes are those which are the children of marked nodes [2]. Corresponding to any marked node u^* (except root), there is a *unique prime* node u', which is its closest prime ancestor. In case u^*'s parent is marked then $u' = u^*$. For every prime node u' with at least one marked node in its subtree, the corresponding closest marked descendant u^* is unique. If u' is marked then the closest marked descendant u^* is same as u'.

Hon et al. [34] showed that, given any node u with u^* being its highest marked descendent (if it exists), the number of leaves in the subtree of u, but not in the subtree of u^* (which we call as fringe leaves) is at most $2g$. This means for a given threshold τ, if z is the number of outputs corresponding to u^* as the locus node, then the number of outputs corresponding to u as the locus is within $z \pm 2g$. This is because of the fact that the number of documents d with $score\big(path(u), d\big) \neq score\big(path(u^*), d\big)$ cannot be more than the number of fringe leaves. Therefore, we maintain the following information at every marked node u^*: the

[2]Note that the number of prime nodes can be $\Theta(n)$ in the worst case.

score of $q-$th highest scored link stabbed by u^* for $q = 1, 2, 4, 8, ...$ By choosing $g = \log n$, the total space can be bounded by $O((n/g) \log n) = O(n)$ words, and can retrieve any particular entry in $O(1)$ time.

Using the above values, the threshold τ corresponding to any given u and k can be computed as follows: first find the highest marked node u^* in the subtree of u ($u^* = u$ if u is marked). Now identify i such that $2^{i-1} < k + 2g \leq 2^i$ and choose τ as the score of 2^i-th highest scored link stabbed by u^*. This ensures that $k \leq z < 2k + O(g) = 2k + O(\log n)$.

3.2.3 Special Structures for Bounded k

In this section, we derive a faster data structures for the case when k is upper bounded by a parameter g. The main idea is to identify smaller sets of $O(g)$ links, such that top-g links stabbed by any node u are contained in one of such sets. Thus by constructing the structure described in Lemma 10 over the links in each such sets, the top-k queries for any $k \leq g$ can be answered faster as follows:

Lemma 11. *There exists a $O(n)$ word data structure for answering top-k queries for $k \leq g$ in $O(\log^2(g/B) + k/B)$ I/O's.*

Recall the definitions of marked nodes and prime nodes from Sec 3.2.2. Let u' be a prime node and u^* (if it exists) be the unique highest marked descendent of u' by choosing a grouping factor g (which will be fixed later). All the links *originated from the subtree* of u' are categorized into the following (see Figure 3.2).

- *near-links:* The links that are stabbed by u^*, but not by u'.
- *far-link:* The links that are stabbed by both u^* and u'.
- *small-link:* The links that are originated from the subtree of u^*, but not stabbed by u^*.
- *fringe-links:* Remaining links. i.e., the links originated not from the subtree of u^*.

Lemma 12. *The number of* fringe-links *and the number of* near-links *of any prime node u' is $O(g)$.*

Proof. The number of leaves in $subtree(u') \backslash subtree(u^*)$ is at most $2g$ [34]. Therefore, the number of *fringe-links* can be bounded by $O(g)$. For every document d whose link originates from $subtree(u^*)$ going out of it ends up as a *near-link* if and only if d exists at one of the leaves of $subtree(u') \backslash subtree(u^*)$. Thus, this can also be bounded by $O(g)$. In the case where u^* does not exist for u', only fringe-links exist. More over the subtree size of u' is $O(g)$ there can be no more than $O(g)$ of these links. □

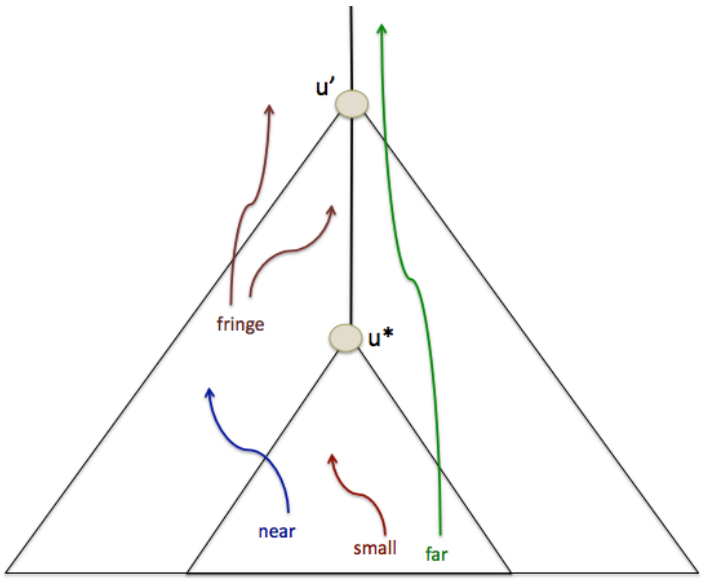

FIGURE 3.2. Categorization of links

Consider the following set, consisting of $O(g)$ links with respect to u': all *fringe-links*, *near-links* and g highest scored *far-links*. We maintain these *links* at u' (as a data structure to be explained later). For any node u, whose closest prime ancestor (including itself) is u', the above mentioned set is called *candidate links* of u. From each u, we maintain the pointer to its closest prime ancestor where the set of *candidate links* is stored.

Lemma 13. *The* candidate links *of any node u contains top-g highest scored links stabbed by u.*

Proof. Let u' be the closest prime ancestor of u. If no marked descendant of u' exist, then all the links are stored as candidate links. Otherwise, *small-links* can never be candidates as they never cross u. Now, if u lies on the path from u' to u^* then all *far-links* will satisfy both origin and target conditions. Else, *far-links* do not qualify. Hence, any link which is not among top-g (highest scored) of these far-links, can never be the candidate. □

Taking a clue from Lemma 12 and 13, for every prime node u', we shall maintain a data structure as in Lemma 10 by considering only the links stored at u', and top-k queries can be answered faster when $k \leq g$. For this we shall define a candidate tree $CT(u')$ of node u' (except the root) to be a modified version of subtree of u' in GST augmented with candidate links stored at u'. Firstly, for every candidate link which is targeted above u', we change the target to v, which will be a dummy parent of u' in $CT(u')$. Now $CT(u')$ consists of those nodes which are either origin or target (after modification) of some candidate link of u'. Moreover, all the nodes in $subtree(u')\backslash subtree(u^*)$ are included as well. Since only the subset of nodes is

selected from $subtree(u')$, our tree is basically a Steiner tree connecting these nodes. Moreover, the tree is edge-compacted so that no degree-1 node remains. Thus, the size of the tree as well as the number of associated links is $O(g)$. Next we do a rank-space reduction of pre-order rank (w.r.t to GST) of the nodes in $CT(u')$ as well as the scores of candidate links.

The candidate tree (no degree-1 nodes) as well as the associated candidate links satisfies all the properties which we have exploited while deriving the structure in Lemma 10. Hence such a structure for $CT(u')$ can be maintained in $O(min(g, size(u'))$ words space and the top-k links in $CT(u')$ stabbed by any node u, with u' being its lowest prime ancestor can be retrieved in $O(\log^2(g/B) + k/B)$ I/O's. The total space consumption of structures corresponding all prime nodes can be bounded by $O(n)$ words as follows: the number of prime nodes with at least a marked node in its subtree is $O(n/g)$, as each such prime node can be associated with a unique marked node. Thus the associated structures takes $O(n/g \times g) = O(n)$ words space. The candidate set of a prime node u' with no marked nodes in its subtree consists of $O(size(u'))$ links, moreover a link cannot be in the candidate set of two such prime nodes. Thus the total space is $O(n)$ words in this case as well. Note that for $g = O(B)$, we need not store any structure on $CT(u')$, because such a candidate tree fits entirely in constant number of blocks which can be processed in $O(1)$ I/Os. This completes the proof of Lemma 11.

3.2.4 I/O-Optimal Data Structure via Bootstrapping

The bounds in Theorem 1 can be achieved by maintaining multiple structures as in Lemma 11. Clearly the structure in Lemma 10 is optimal for $k \geq B\log^2(n/B)$. However, for handling the case when $k < B\log^2(n/B)$, we shall choose the grouping factor $g_i = B(\log^{(i)}(n/B))^2$, for $i = 1, 2, 3, .., h \leq \log^* n$ and maintain h separate structures as in Lemma 11, occupying $O(nh)$ space. Thus top-k query for any $k \geq g_h$ can be answered by querying on the structure corresponding to the grouping factor g_j, where $g_j \geq k > g_{j+1}$ in $O(\log^2(g_j/B) + k/B) = O(g_{j+1}/B + k/B) = O(k/B)$ I/Os. For $k < g_h$, we shall query on the structure corresponding to the grouping factor g_h, and the I/Os are bounded by $O(\log^2(g_h/B) + k/B) = O(\log^{(h)} n + k/B)$. This completes the proof of Theorem 1.

3.3 Adapting to Internal Memory

Our framework can also be used for improving the existing internal memory results. Specifically, we shall derive a linear space data structure for retrieving top-k documents in $O(k)$ time, once the locus of the

pattern match is given. This result improves the previous work [34, 48] by eliminating the additive term p. In applications like cross-document pattern matching [39], the locus can be computed in much faster $O(\log\log p)$ time than $O(p)$ [39]. In many pattern matching applications, for example in suffix-prefix overlap, maximal substring matches, and autocompletion search (like in Google Instant^{TM}) multiple loci are searched with amortized constant time for each locus. In such situations, having extra $O(p)$ as in [34, 48] leads to inefficient solutions, where as our data structure can support faster queries. The result is summarized as follows:

Theorem 2. *There exists an $O(n)$ word space data structure in word RAM model for solving (sorted) top-k document retrieval problem in $O(k)$ time, once the locus of the pattern match is given.*

Proof. Our external memory framework can be adapted to internal memory by choosing $B = \Theta(1)$, and by replacing the external memory substructures by the corresponding internal memory counterparts. Retrieving the outputs among high-ranked links is reduced to $O(\log n)$ 3-sided range reporting queries. By using an interval tree like approach, the problem of retrieving outputs among equi-ranked links also can be reduced to $O(\log n)$ 3-sided range reporting queries. By using the linear-space sorted range reporting structure by Brodal et al. [8] for 3-sided range reporting, the outputs can be obtained in the sorted order of score. Further, these sorted outputs from $O(\log n)$ different places can be merged using an atomic heap [21], which is capable of performing all heap operations in $O(1)$ time, provided the number of elements in the heap is $O(\log^{O(1)} n)$ as in our case. At the beginning of each of these $O(\log n)$ queries, we may need to perform a binary search for finding the boundaries, thus resulting in a total query time of $O(\log^2 n + k)$, which is $O(k)$ for $k \geq \log^2 n$. The space can be bounded by $O(n)$ words. For the case when $k < \log^2 n$, we obtain a linear space and $O(\log^2 \log n + k)$ query time structure by using the ideas from Sec 3.2.3 (here we choose grouping factor $g = \log^2 n$). Again, this structure can answer queries in $O(k)$ time for $k \geq \log^2 \log n$. We do not continue this bootstrapping further. Instead, we make use of the following observation: the candidate set of a node consists of only $O(g)$ links, hence a pointer to any particular link within the candidate set of any node can be maintained in $O(\log g) = O(\log\log n)$ bits. Thus, at every node u in GST, we shall maintain the top-$(\log^2 \log n)$ links stabbed by u in the decreasing order of score as a pointer to its location within the candidate set of u. This occupies $O(n \log^3 \log n)$ bits or $o(n)$ words and top-k queries for any $k \leq \log^2 \log n$ can be answered in $O(k)$ time by chasing the first k pointers and retrieving the documents associated with the corresponding links. This completes the proof of Theorem 2. □

Chapter 4
Succinct Space Data Structures

Data structures occupying space close to the minimum space needed for maintaining the data is referred as compressed/succinct data structures. Although, this compression comes with the cost of higher query time (theoretically), in practice these data structures may fit in faster memory and small memory devices, while uncompressed data may not. The generalized suffix tree (GST) data structure can be replaced by its compressed counter part. However, the challenge is to maintain the augmented information in compressed form. In this section, we present a compressed space data structure of space $2|\mathsf{CSA}^*| + D\log\frac{n}{D} + O(D) + o(n)$ bits and query time $O(t_s(p) + k \times t_{sa} \log k \log^{\epsilon} n)$ for the case where $score(\cdot,\cdot) = \text{TF}(\cdot,\cdot)$. Here $|\mathsf{CSA}^*|$ denotes the maximum space (in bits) to store either a compressed suffix array (CSA) of the concatenated text with all the given documents in $\mathcal{D}$, or all the CSAs of individual documents, t_{sa} is the time decoding a suffix array value, $t_s(p)$ is the time for computing the suffix range of P using CSA, and $\epsilon >$ is any constant.

4.1 Related Work

Let $\mathsf{T} = d_1 d_2 d_3 \cdots d_D$ be the text (of n characters from an alphabet set $\Sigma = [\sigma]$) obtained by concatenating all the documents in $\mathcal{D}$. Recall that the last character of each document is \$, a special symbol that does not appear anywhere else in T. For succinct data structures (which take space close to the size of T in its compressed form), existing work focussed on the case where the relevance metric is term frequency or static importance score. Most of the succinct data structures used a key idea from an earlier paper by Sadakane [62], where he showed how to compute the TF-IDF score of each document, by maintaining a compressed suffix array CSA of T along with a compressed suffix array CSA_r of each document d_r (see Section 2.3 for the definition of CSA).

The first succinct space data structure for answering top-k frequent document retrieval queries was proposed by Hon et al. [34]. Their data structure occupies $2|\mathsf{CSA}^*| + o(n) + D\log\frac{n}{D} + O(D)$ bits of space and answers a query in $O(t_s(p) + k \times t_{sa} \log^{3+\epsilon} n)$ time. While retaining this space, the query time is improved to $O(t_s(p) + k \times t_{sa} \log D \log(D/k) \log^{1+\epsilon} n)$ by Gagie et al. [23]. Belazzougui et al. [6] improved this further to $O(t_s(p) + k \times t_{sa} \log k \log(D/k) \log^{\epsilon} n)$. Our result of Theorem 4 in this paper (initially appeared in [35]) achieves an even faster query time of $O(t_s(p) + k \times t_{sa} \log k \log^{\epsilon} n)$. An

open problem of designing a space-optimal data structure is positively answered by Tsur [64], where he proposed an $|\mathsf{CSA}| + o(n) + O(D) + D\log(n/D)$-bit data structure with $O(t_s(p) + k \times t_{sa} \log k \log^{1+\epsilon} n)$ query time; very recently, Navarro and Thankachan [53] improved the time to $O(t_s(p) + k \times t_{sa} \log^2 k \log^{\epsilon} n)$. Top-$k$ *important* document retrieval (i.e., the score function is document importance) is also a well-studied problem, and the best known succinct data structure appeared in [6]. This data structure takes $|\mathsf{CSA}| + o(n) + O(D) + D\log(n/D)$ bits of space, and answers a query in $O(t_s(p) + k \times t_{sa} \log k \log^{\epsilon} n)$ time. See Table 4.1 for comparison purpose, where $\log D$ and $\log(D/k)$ simplified to the worst-case bound of $\log n$ in the reporting time and $\sigma \leq D$ is assumed as $o(n/\log n)$.

TABLE 4.1. Comparison Table

Source	**Space (in bits)**	**Per-Document Reporting Time**
Hon et al. [34]	$2\|\mathsf{CSA}^*\| + o(n)$	$O(t_{sa} \log^{3+\epsilon} n)$
Gagie et al. [23]	$2\|\mathsf{CSA}^*\| + o(n)$	$O(t_{sa} \log^{3+\epsilon} n)$
Belazzougui et al. [6]	$2\|\mathsf{CSA}^*\| + o(n)$	$O(t_{sa} \log k \log^{1+\epsilon} n)$
Ours (Theorem 4)	$2\|\mathsf{CSA}^*\| + o(n)$	$O(t_{sa} \log k \log^{\epsilon} n)$
Tsur [64]	$\|\mathsf{CSA}\| + o(n)$	$O(t_{sa} \log k \log^{1+\epsilon} n)$
Navarro and Thankachan [53]	$\|\mathsf{CSA}\| + o(n)$	$O(t_{sa} \log^2 k \log^{\epsilon} n)$

4.2 Our Data Structure

We start with the following notation:

- $Leaf(x)$ denotes the set of leaves in the subtree of node x in GST.
- $Leaf(x \backslash y) = Leaf(x) \setminus Leaf(y)$, the leaves in the subtree of x, but not in that of y.

Let g be a parameter called the *grouping factor*. Using the following scheme, we identify a subset S_g of nodes, called *marked nodes*, in GST: First, we traverse the leaves of GST from left to right to form groups of g contiguous leaves. That is, the first group consists of leaves $\ell_1, \ell_2, \ldots, \ell_g$, the next group consists of $\ell_{g+1}, \ldots, \ell_{2g}$, and so on. In total, there will be $\lceil n/g \rceil$ groups. Next, for each group, we mark the lowest common ancestor (lca) in GST of its first and last leaves; the total number of marked nodes will be at most $\lceil n/g \rceil$. After that, we do further marking, such that if nodes u and v are marked, then $lca(u, v)$ will be marked. Finally, we mark the leftmost and the rightmost leaves within the subtree rooted at each marked node.

Lemma 14. *The above marking scheme ensures the following properties:*

- *The number of marked nodes, $|S_g|$, is bounded by $O(n/g)$.*
- *If there is no marked node in the subtree of x, then $|Leaf(x)| < 2g$.*
- *The highest marked descendant node y of any unmarked node x, if it exists, is unique, and $|Leaf(x \backslash y)| < 2g$.*

Proof. The number of groups at the end of first step is $\lceil n/g \rceil$, and at most one internal node corresponding to each group is marked. Thus, at the end of the first step, there are at most $\lceil n/g \rceil$ marked nodes. Next, we mark the *lca* of these marked nodes; the total number of marked nodes will at most be doubled (as the marked nodes now form an induced subtree, with marked nodes at the end of first step as leaves), so that it is bounded by $O(n/g)$. Finally, we mark the leftmost and the rightmost leaf nodes of every marked node. Thus, the the total number of marked nodes will at most be tripled, so that it is bounded by $O(n/g)$. This gives the result in (1).

Whenever $|Leaf(x)| \geq 2g$, there will be at least one group completely contained in the subtree of x. The *lca* of the first and the last leaves in such a group is within the subtree of x, and is marked. Thus, by contraposition, the result in (2) follows.

The last statement in the lemma can be proved as follows: Let ℓ_L and ℓ_R be the leftmost and the rightmost leaves in the subtree of x. Then, according to our marking scheme, y is the *lca* of leaves $\ell_{L'}$ and $\ell_{R'}$, where $L' = g\lceil L/g \rceil + 1$ and $R' = g\lceil R/g \rceil$. Let ℓ_{L^*} and ℓ_{R^*} be the leftmost and the rightmost leaves respectively, that are in the subtree of y. Then clearly $L \leq L^* \leq L' < L + g$ and $R \geq R^* \geq R' > R - g$. Therefore, $|Leaf(x \backslash y)| = (L^* - L) + (R - R^*) < 2g$. □

Let $\mathsf{top}(x, k)$ represent the list (or set) of top-k documents corresponding to a pattern with node x as the locus. Maintaining $\mathsf{top}(x, k)$ explicitly for all possible x values and k values is not possible in compressed space. Instead, we maintain $\mathsf{top}(x, k)$ only for marked nodes x (with respect to various carefully chosen g values) and for values of k that are powers of 2, such that $\mathsf{top}(x, k)$ for the general x and k can be efficiently computed on the fly. We next prove the following lemma.

Lemma 15. *By maintaining a data structure called GST_g of size $O((n/g) \log g) + O(n/\log^2 n)$ bits, the following query can be answered in $O(1)$ time: Given a suffix range $[sp, ep]$ of a pattern P as an input, find the node v_P^* and the range $[sp^*, ep^*]$, where* `(i)` *v_P^* denotes the highest-marked descendent of the locus node*

v_P of P, and (ii) *ℓ_{sp^*} and ℓ_{ep^*} denote, respectively, the leftmost leaf and the rightmost leaf in the subtree of v_P^*.*

Proof. The data structure GST_g, requiring $O((n/g)\log g) + O(n/\log^2 n)$ bits of space, consists of the following components:

- A compact trie obtained by retaining only those nodes in GST that are marked. Then, corresponding to every marked node in GST, there will be a unique node in this trie and vice versa. As the number of marked nodes is $O(n/g)$, the topology of this trie can be maintained in $O(n/g)$ bits of space (refer to Section 2.5).

- A bit-vector $\mathsf{B_{no}}[1..2n]$, where $\mathsf{B_{no}}[i] = 1$ if the ith node in GST is marked, else 0. This can be maintained in $|S_g|\log(n/|S_g|) + O(|S_g|) + O(n/\log^{O(1)} n) = O((n/g)\log g) + O(n/\log^2 n)$ bits of space [60], so that the operations $select_{\mathsf{B_{no}}}(j)$ (the position of the jth 1 in $\mathsf{B_{no}}$) and $rank_{\mathsf{B_{no}}}(i)$ (the number of 1s in $\mathsf{B_{no}}[1..i]$) can be supported in $O(1)$ time.

- A bit-vector $\mathsf{B_{le}}[1..n]$, where $\mathsf{B_{le}}[i] = 1$ if the ith leftmost leaf in GST is marked, else 0. As in the case of $\mathsf{B_{no}}$, $\mathsf{B_{le}}$ will be maintained in $O((n/g)\log g) + O(n/\log^2 n)$ bits, so that it can support $select_{\mathsf{B_{le}}}(\cdot)$ and $rank_{\mathsf{B_{le}}}(\cdot)$ operations in $O(1)$ time.

Given an input suffix range $[sp, ep]$, the sp^*th leaf is the first marked leaf towards the right side of ℓ_{sp} (inclusive), and the ℓ^*_{ep}th leaf is the last marked leaf towards the left side of ℓ_{ep} (inclusive), in GST. These two leaves will correspond to the sp'th and the ep'th leaves in the compact trie, where

$$sp' = 1 + rank_{\mathsf{B_{le}}}(sp-1) \text{ and } ep' = rank_{\mathsf{B_{le}}}(ep);$$

the desired values of sp^* and ep^* can thus be computed, in $O(1)$ time, by $sp^* = select_{\mathsf{B_{le}}}(sp')$ and $ep^* = select_{\mathsf{B_{le}}}(ep')$.

We now show how to find v_P^*, which is the lca of ℓ_{sp^*} and ℓ_{ep^*} in GST. As GST is not stored explicitly, we shall find v_P^* in an indirect way. First, we identify the leaf nodes corresponding to ℓ_{sp^*} and ℓ_{ep^*} in the compact trie, which is its sp'th and ep'th leaves. Next, we find their lca (say, with preorder rank x) in the compact trie; such a node will correspond to v_P^* in GST. It follows that v_P^* is the xth marked node in GST, so that we can finally find (the preorder rank of) v_P^* in GST by $select_{\mathsf{B_{no}}}(x)$. The procedure again takes $O(1)$ time in total, as it involves only a constant number of rank/select operations and an lca operation. □

4.2.1 The Compressed Data Structure

Our compressed data structure will make use of both CSA of the concatenated text T of all the documents, and a compressed suffix array CSA_r of each individual document d_r. We prove the following in this section.

Theorem 3. *A given collection $\mathcal{D}$ of D documents with n characters in total taken from an alphabet set $\Sigma = [\sigma]$, we can build a data structure of space $2|\mathsf{CSA}^*| + D \log \frac{n}{D} + O(D) + o(n)$ bits, such that whenever a pattern P (of p characters) and an integer k come as a query, the data structure returns those k documents with the highest $\text{TF}(P, \cdot)$ values in decreasing order of $\text{TF}(P, \cdot)$ in $O(t_s(p) + k \times t_{sa} \log^{2+\epsilon} n)$ time; here, $|\mathsf{CSA}^*|$ denotes the maximum space (in bits) to store either a compressed suffix array (CSA) of the concatenated text with all the given documents in $\mathcal{D}$, or all the CSAs of individual documents, t_{sa} is the time decoding a suffix array value, $t_s(p)$ is the time for computing the suffix range of P using CSA, and $\epsilon >$ is any constant.*

A set $S_{\mathsf{cand}} \subseteq \mathcal{D}$ is called a *candidate set* of a query if it is a multiset that contains all those documents in the answers to the query. Therefore, once the candidate set is given, the top-k query can be answered by first finding the $\text{TF}(P, d_r)$ score of each document $d_r \in S_{\mathsf{cand}}$, and then reporting the k highest-scoring ones.

Lemma 16. *Once the candidate set S_{cand} is identified, a top-k query can be answered using CSA and the structure described in Lemma 2 in time $O(|S_{\mathsf{cand}}| \times t_{sa} \log \log n + k \log k)$.*

Proof. First, we remove duplicates in S_{cand} if there are any. This can be easily done in $O(|S_{\mathsf{cand}}|)$ time by maintaining an extra bit vector $\mathsf{B}_{\mathsf{cand}}[1..D]$, where all its bits are initialized to 0. Note that this additional structure will not change the space bound in Theorem 3. Then, we scan all documents in S_{cand} one by one and do the following: If a document $d_r \in S_{\mathsf{cand}}$, then we check if $\mathsf{B}_{\mathsf{cand}}[r]$ is 0. If so, we set $\mathsf{B}_{\mathsf{cand}}[r] = 1$; otherwise, we delete such an occurrence of d_r (which is a duplicate) from S_{cand}. After scanning all the documents in S_{cand}, we can reset all bits in $\mathsf{B}_{\mathsf{cand}}$ back to 0 by rescanning S_{cand} once.

Next, we compute the $\text{TF}(P, d_r)$ score for all those documents $d_r \in S_{\mathsf{cand}}$ in $O(t_{sa} \log \log n)$ time per document (refer to Lemma 3). To retrieve the top-k answers from this, we first find the score X of the kth highest-scoring document using the linear time selection algorithm [7]. Then, we get those documents whose scores are at least X; note that there may be more than k of them, because of ties. To get the desired answer, we shall remove the excess (whose scores are equal to X). Finally, we spend another $O(k \log k)$ time to obtain the answers in sorted order of their scores. □

The query time in the above lemma is dependent on the size $|S_{\mathsf{cand}}|$ of the candidate list. To speed up the whole process (so as to achieve the claimed result in Theorem 3), our objective is to find a candidate set whose size is as small as possible.

Data Structure for Top-k Queries for a Fixed k. First, we define a data structure for answering top-k frequent queries, where k is fixed in advance. The data structure consists of (i) a compressed suffix array CSA of T; (ii) the document array E (represented in $|\mathsf{CSA}^*| + D \log \frac{n}{D} + O(D) + o(n)$ bits, refer to Lemma 2); (iii) an auxiliary structure that includes (a) GST_g (refer to Lemma 14) with a grouping factor $g = k \log^{2+\epsilon} n$, and (b) for each marked node $x \in S_g$ in GST, we store $\mathsf{top}(x, k)$ explicitly in $k \log D$ bits. The total space of the auxiliary structures is $O((n/g)k \log D) + O(n/\log^2 n) = o(n/\log n)$ bits.

To answer the query, we first find the suffix range $[sp, ep]$ of P in $t_s(p)$ time using CSA. Let v_P be the locus node of P. Then, we find v_P^* and $[sp^*, ep^*]$ in $O(1)$ time, where v_P* is the highest marked descendent of v_P (if it exists), and $[sp^*, ep^*]$ is the suffix range corresponding to v_P^* in GST (refer to Lemma 15). Then,

$$\mathsf{top}(v_P^*, k) \cup \left\{ d_{E\,[j]} \mid j \in [sp, sp^* - 1] \cup [ep^* + 1, ep] \right\}$$

will be a candidate set.[1] The number of documents in $\mathsf{top}(v_P^*, k)$ is at most k, and the number of remaining documents in the candidate set is at most $2g$ (refer to Lemma 14). To construct the candidate set, we first retrieve all documents in $\mathsf{top}(v_P^*, k)$ in $O(k)$ time, as these documents are precomputed and explicitly stored at v_P^*; then, since each $E\,[\cdot]$ value can be decoded in $O(t_{sa})$ time (refer to Lemma 2), we retrieve all the remaining documents in $O(g \times t_{sa})$ time. In summary, we obtain a candidate set of $O(g + k)$ documents in $O(g \times t_{sa} + k)$ time. Combining with Lemma 16, the top-k documents can be answered in another $O((g + k) \times t_{sa} \log\log n)$ time. By substituting $g = k \log^{2+\epsilon} n$ the resulting query time will be $O(t_s(p) + k \times t_{sa} \log^{2+\epsilon} n \log\log n) = O(t_s(p) + k \times t_{sa} \log^{2+\epsilon} n)$ (the $\log\log n$ term is absorbed in the $\log^{\epsilon} n$ term).

Data Structure for Top-k Queries for General k. To support top-k queries for general k, we maintain CSA, E , and (at most) $\log D$ auxiliary structures for any fixed k that is a power of 2 (i.e., $k = 1, 2, 4, 8, \ldots, D$). Since an auxiliary structure for a specific k requires $o(n/\log n)$ bits, the overall increase

[1] In the boundary case where v_P^* does not exist, the candidate set is simply $\{d_{E\,[j]} \mid j \in [sp, ep]\}$, whose size is at most $2g$ (refer to Lemma 14).

in total space is bounded by $o(n)$ bits. Now, a top-k query for a general k can be answered by choosing $z = 2^{\lceil \log_2 k \rceil}$ and retrieving the top-z documents by querying on the auxiliary structure specific to z. Then, we select the k highest-scoring documents (using [7]) and report them in decreasing order of score. Since $k = \Theta(z)$, the resulting query time will be $O(t_s(p) + k \times t_{sa} \log^{2+\epsilon} n)$. This completes the proof of Theorem 3.

As a corollary, we can obtain a simple compact data structure by re deriving Theorem 3 with $g = z \log^{1+\epsilon} D$, and maintaining E explicitly as in Lemma 1.

Lemma 17. *There exists a data structure of size $|\mathsf{CSA}| + n \log D(1 + o(1))$ bits for the top-k frequent document retrieval problem with $O(t_s(p) + k \log^{1+\epsilon} D)$ query time, where $\epsilon >$ is any constant.*

4.2.2 Faster Compressed Data Structure

This section describes how to improve the data structure to speed up the query. The idea is to choose a smaller grouping factor, thereby reducing the size of the candidate set. However, this will result in more marked nodes, so that explicit storage of precomputed answers (with $\log D$ bits per entry) at these marked nodes will lead to a non-succinct solution. Our key contribution is to show how these precomputed lists can be encoded in $O(\log \log n)$ bits per entry. Our main result is summarized as follows.

Theorem 4. *A given collection $\mathcal{D}$ of D documents with n characters in total taken from an alphabet set $\Sigma = [\sigma]$, we can build a data structure of $2|\mathsf{CSA}^*| + D \log \frac{n}{D} + O(D) + o(n)$ bits of space, such that whenever a pattern P (of p characters) and an integer k come as a query, the data structure returns those k documents with the highest $\mathrm{TF}(P, \cdot)$ values in decreasing order of $\mathrm{TF}(P, \cdot)$ in $O(t_s(p) + k \times t_{sa} \log k \log^{\epsilon} n)$ time; here, $|\mathsf{CSA}^*|$ denotes the maximum space (in bits) to store either a compressed suffix array (CSA) of the concatenated text with all the given documents in $\mathcal{D}$, or all the CSAs of individual documents, t_{sa} is the time decoding a suffix array value, $t_s(p)$ is the time for computing the suffix range of P using CSA, and $\epsilon >$ is any constant.*

Data Structure for Top-k Queries for a Fixed k. Similar to the data structure in previous section, we define a data structure for answering top-k frequent queries, where k is fixed in advance. The data structure consists of (i) a compressed suffix array CSA; (ii) the document array E (represented in $|\mathsf{CSA}^*| + D \log \frac{n}{D} + O(D) + o(n)$ bits, refer to Lemma 2) (iii) an auxiliary structure with respect to two grouping factors g and h, which is defined as follows. First, we mark the nodes in GST based on two grouping factors g and h, where

$g = k \log^{2+\epsilon} n$ and $h = k \log k \log^{\epsilon} n$. Then, we maintain the corresponding GST_g and GST_h in a total of $O((n/g) \log g + (n/h) \log h) = o(n/k)$ bits (refer to Lemma 15).

In order to distinguish the marked nodes based of these two different grouping factors, we shall use the following terminology: If a node is marked as per the grouping factor g, we shall simply call it a marked node. Otherwise, if a node is marked as per the grouping factor h only, we shall call it as a *prime* node (see Figure 4.1).

We now describe the query answering algorithm. Let v_P be the locus node of the input pattern P in GST with v'_P and v^*_P, respectively, being its highest prime descendant and highest marked descendant (if they exist). Let $[sp, ep]$, $[sp', ep']$, and $[ep^*, ep^*]$, respectively, be the ranges of leaves within the subtree of v_P, v^*_P and v'_P. Note that the following inequalities hold (refer to Lemma 14):

- $sp \leq sp' \leq sp^* \leq ep^* \leq ep' \leq ep$;
- $sp' - sp < h$ and $ep - ep' < h$;
- $sp^* - sp' < g$ and $ep' - ep^* < g$.

Then,

$$\mathsf{top}(v'_P, k) \cup \left\{ d_{E\,[j]} \mid j \in [sp, sp' - 1] \cup [ep' + 1, ep] \right\}$$

will be a candidate set, where we shall denote it by S^h_{cand}. The number of documents in $\mathsf{top}(v'_P, k)$ is at most k, and the number of the remaining documents in the candidate set is at most $2h$.

Once S^h_{cand} is given, it takes only an extra $O((h + k) \times t_{sa} \log \log n) = O(k \times t_{sa} \log k \log^{\epsilon} n)$ time for answering a top-k query (using Lemma 16). Note that the documents $d_{E\,[j]}$ for $j \in [sp, sp' - 1] \cup [ep' + 1, ep]$ can be computed on the fly in $O(h \times t_{sa})$ time, which will not affect the overall time complexity. It remains to show how to obtain the list $\mathsf{top}(v'_P, k)$ efficiently. By the following lemma, the total query time can be bounded by $O(t_s(p) + k \times t_{sa} \log k \log^{\epsilon} n)$.

Lemma 18. *In $O(n/(\log k \log^{\epsilon} n)) + o(n/\log n)$ bits of space, we can encode $\mathsf{top}(\cdot, k)$ corresponding to every prime nodes, such that $\mathsf{top}(w', k)$ of any prime node w' can be decoded in $O(k \times t_{sa} \log \log n)$ time.*

Proof. We shall give an encoding of $\mathsf{top}(w', k)$ for each prime node w' that allows us to obtain a candidate set corresponding to w' as the locus. Then, by using Lemma 16, we can compute the desired $\mathsf{top}(w', k)$ based on the candidate set.

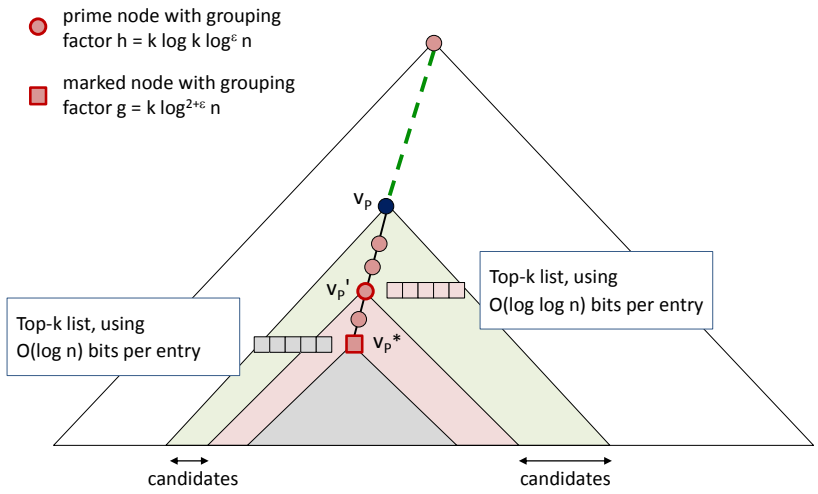

FIGURE 4.1. Query answering with prime nodes and marked nodes

Let w^* be the highest marked descendent of w' (if it exists). Let $[L', R']$ and $[L^*, R^*]$, respectively, denote the range of leaves in the subtree of w' and w^*. A candidate set corresponding to w' as the locus (i.e., a superset of $\mathsf{top}(w', k)$) is given by

$$\mathsf{top}(w^*, k) \cup \left\{ d_{E\,[j]} \mid j \in [L', L^* - 1] \cup [R^* + 1, R'] \right\}.$$

The set $\mathsf{top}(w^*, k)$ can be obtained in $O(k)$ time by maintaining $\mathsf{top}(\cdot, k)$ for each marked node explicitly, which requires a total of $O((n/g)k \log D) = o(n/\log n)$ bits. For the set $\{d_{E\,[j]} \mid j \in [L', L^* - 1] \cup [R^* + 1, R']\}$ of the remaining documents, we select only the subset of its top k documents; then we see that this subset, when combined with $\mathsf{top}(w^*, k)$, still forms a candidate set corresponding to w' as the locus. In other words, even though we have $O(g)$ documents in this category, only at most k of them can be among $\mathsf{top}(w', k)$. Now, suppose that these k documents can be encoded in $O(k \log\log n)$ bits, while supporting decoding in $O(k \times t_{sa})$ time. Thus, the total space for all the encodings in all the prime nodes is $O(n/(\log k \log^{\epsilon} n))$ bits, and we can obtain the desired candidate set in a total of $O(k \times t_{sa})$ time. Consequently, $\mathsf{top}(w', k)$ can be computed in $O(k \times t_{sa} \log\log n)$ time using Lemma 16.

It remains to show how to encode the selected top k documents with the claimed performance. For each such document d_j, it can be associated with an integer $i \in [L', L^* - 1] \cup [R^* + 1, R']$ such that $E[i] = j$. If we replace each such i by its relative position in $[L', L^* - 1] \cup [R^* + 1, R']$, this problem can be rephrased as the encoding of k distinct integers drawn from $[1, 2g]$. An encoding with $O(k \log\log n)$ bits of space and $O(k)$ decoding time can be achieved, by maintaining a bit vector $\mathsf{B}_{w',k}$ with constant-time *select* operations

supported [61]; here, $\mathsf{B}_{w',k}[1..2g]$ is defined such that $\mathsf{B}_{w',k}[i] = 1$ if and only if i is an integer to be stored. Therefore $\mathsf{B}_{w',k}$ can be maintained in $k\log(2g/k) + O(k) = O(k\log\log n)$ bits of space, and the stored integers can be decoded by $select_{\mathsf{B}_{w',k}}(j)$ queries for $j = 1, 2, 3, \ldots, k$. Finally, given these integers (relative positions), the corresponding document can be retrieved in $O(t_{sa})$ time. This completes the proof. □

Putting everything altogether, we have the following lemma.

Lemma 19. *The auxiliary structure for a specific k takes $O(n/(\log k \log^{\epsilon} n)) + o(n/\log n) + o(n/k)$ bits of space. Given the suffix range $[sp, ep]$ of a pattern P, a top-k frequent document retrieval query can be answered in $O(k \times t_{sa} \log k \log^{\epsilon} n)$ time.*

Data Structure for Top-k Queries for General k. To support top-k queries for general k, we maintain CSA, E , and (at most) $\log D$ auxiliary structures for $k = 1, 2, 4, 8, \ldots, D$, analogous to how we handle the general k case as in previous section. This requires a total of

$$\sum_{z=1,2,4,\ldots,D} \Big(O(n/(\log^{\epsilon} n \log z)) + o(n/\log n) + o(n/z)\Big) = o(n) \text{ bits.}$$

A top-k query can be answered by choosing $z = 2^{\lceil \log_2 k \rceil}$ and retrieving the top-z documents by querying on the auxiliary structure specific to z. Then, we select the k highest-scoring documents (using [7]) and report them in decreasing order of score. Combining with the fact that $k = \Theta(z)$, we obtain Theorem 4.

4.3 Extensions

Although we described our result in terms of term frequency as the scoring function, we can in fact extend it to some other scoring functions that are succinctly calculable. Unfortunately, we do not know if $\mathrm{TP}(\cdot,\cdot)$ is succinctly calculable. In contrast, $docrank(\cdot,\cdot)$ is not only succinctly calculable, but is trivial to compute. In fact, to support top-k queries with the $docrank$ metric, we do not even need the document array E using Lemma 2, but only the bit vector B_E and an array R of size $D\log D$ bits such that $\mathsf{R}[r]$ gives the relative $docrank$ of document d_r among the others; after the change, we can still compute $docrank$ of any document $d_{E[i]}$ within the same time bound. This gives the following theorem.

Theorem 5. *A given collection $\mathcal{D}$ of D documents with n characters in total taken from an alphabet set $\Sigma = [\sigma]$, we can build a data structure of $|\mathsf{CSA}| + o(n) + D\log\frac{n}{D} + O(D) + D\log D$ bits of space, such*

that whenever a pattern P (of p characters) and an integer k come as a query, the data structure returns those k documents with the highest $docrank(\cdot)$ values in decreasing order of $docrank(\cdot)$ in $O(t_s(p) + k \times t_{sa} \log k \log^{\epsilon} n)$ time; here, $docrank(d_r)$ of a document d_r is a static importance score associated with d_r, $t_s(p)$ is the time to search for a pattern of length p with CSA*, t_{sa} is the time to compute a suffix array entry with* CSA*, and $\epsilon >$ is any constant.*

See [4] for a similar result, which appeared earlier but used different techniques.

Chapter 5
Compact Space Data Structures

In this section [1], we show that it is possible to get very close to optimal time within compact space. We prove the following result (appeared in [54]), where we remark that the top-k results are not returned in sorted order of relevance.

Theorem 6. *There exists a compact index of* $|\mathsf{CSA}| + n \log D + o(n \log D)$ *bits and near-optimal* $O(p + k \log^* n)$ *query time time, for the (unsorted) top-*k *frequent document retrieval problem, where* $\log^* n$ *is the iterated logarithm of* n*.*

In Section 5.5 we achieve $O(p + k \lg^* k)$ time, using $o(n \lg \sigma)$ further bits.

5.1 Related Work

The *document array* (refer to Section 2.6 for the definition) is a powerful data structure for solving string retrieval problems, and its space occupancy is $n\lceil \log D \rceil$ bits. This was first introduced in [65] for solving the document listing problem. Later, Culpepper et al. [14] showed how to efficiently handle top-k frequent document retrieval queries using a simple data structure, which is essentially a wavelet tree maintained over the document array. Although their query algorithm is only a heuristic (no worst-case bound), it works well in practice, with space occupancy roughly $1 - 3$ times the text size. From now onwards, a data structure that allows a space term of roughly $n \log D$ bits, corresponding to the document array, will be called a *compact* data structure. Gagie et al. [23] proposed two compact data structure of sizes $|\mathsf{CSA}|+n \log D(1+o(1))$ bits and $|\mathsf{CSA}| + O(n \log D/ \log \log D)$ bits, with query time bounds of $O(t_s(p) + k \times \log D \log(D/k) \log^\epsilon n)$ and $O(t_s(p) + k \times t_{sa} \log D \log(D/k) \log^\epsilon n)$, respectively. Belazzougui et al. [6] showed that the $\log D$ factor in the query time of both results by Gagie et al. can be converted to $\log k$ without increasing the space; they also showed a data structure of size $|\mathsf{CSA}| + O(n \log \log \log D)$ bits with $O(t_s(p) + k \times t_{sa} \log k \log^{1+\epsilon} n)$ query time. Hon et al. [33, 30] proposed another compact space data structure of space $|\mathsf{CSA}| + 2n \log D(1 + o(1))$ bits with query time $O(p + \log^4 \log n + k(\log \log n + \log k))$. In the same paper, they proposed an even

[1] This section previously appeared as, Gonzalo Navarro and Sharma V. Thankachan, *Top-k Document Retrieval in Compact Space and Near-Optimal Time*, Proceedings of International Symposium on Algorithms and Computation (ISAAC), 2013, pages 394–404. It is reprinted by permission of Springer. also see [55] for its journal version.

more space efficient of space $|\mathsf{CSA}| + n \log D(1 + o(1))$ bits, however the query time is increased to $O(p + \log^6 \log n + k((\log \log n \log \sigma)^{1+\epsilon} + \log k))$. Navarro and Nekrich [48] gave a data structure of size $O(n(\log \sigma + \log D))$ bits data structure with optimal $O(p + k)$ time; however, the hidden constants within the big-O notations are not small in practice [38]. It has been shown that, compact space data structures provide the best practical performance [38, 14] compared to linear space data structures [58] (which are less efficient in terms of space occupancy) and the succinct space data structures [15, 49] (which are less efficient in terms of query processing time).

5.2 The Data Structure

Three main components of our structure are a generalized suffix tree (GST), the document array, and some precomputed answer lists. These are described next. We redefine them for the completeness of this section.

Document Array (E). Define a bit-vector $B[1..n]$, such that $B[i] = 1$ iff $T[i] = \$$. Then suffix $T[i, n]$ belongs to document d_r if $r = 1 + rank_B(i)$, where $rank_B(i)$ is the number of 1s in $B[1, i]$. The document array $E\,[1..n]$ is defined as $E\,[j] = r$ if the suffix $\mathsf{SA}[j]$ belongs to document d_r. Moreover, we say that the corresponding leaf node ℓ_j is *marked* with document d_r. Now,

- $rank_E\,(r, i)$ returns the number of occurrences of r in $E\,[1, i]$;
- $select_E\,(r, j)$ returns i where $E\,[i] = r$ and $rank_E\,(r, i) = j$; and
- $access_E\,(i)$ returns $E\,[i]$;

Then we have use the following representation for E [5].

Lemma 20. *The document array E can be stored in $n \log D + o(n \log D)$ bits and support queries $rank_E$, $select_E$ and $access_E$ in times $O(\log \log n)$, $O(f(n, D))$ and $O(1)$ respectively, where $f(n, D) = \omega(1)$ is any non-constant function.*

The so-called *partial rank* query can be added to this repertoire [5].

Lemma 21. *Operation $rank_E\,(E\,[i], i)$ can be supported in constant time by storing $O(n \lg \lg D) = o(n \log D)$ additional bits on top of the E .*

Thus the total space of this component is $n \log D + o(n \log D)$ bits.

Precomputed Answer Lists. We start with the following definitions:

- $Leaf(x)$ is the set of leaves in the subtree of node x in GST.
- $Leaf(x \backslash y) = Leaf(x) \setminus Leaf(y)$, the leaves in the subtree of x, but not in that of y.
- $score(r, x)$ is the number of leaves in $Leaf(x)$ marked with document d_r (i.e., $|\{\ell_i \in L(x), E[i] = r\}|$).

Now using the marking scheme described in Section 4.2, we identify the set S_g of marked nodes in GST based on the grouping parameter g. Notice that the following properties are ensured (refer to Lemma 14).

- The number of marked nodes, $|S_g|$, is bounded by $O(n/g)$.
- If there is no marked node in the subtree of x, then $|Leaf(x)| < 2g$.
- The highest marked descendant node y of any unmarked node x, if it exists, is unique, and $|Leaf(x \backslash y)| < 2g$.

Let $\mathsf{top}(x, k)$ represent the list (or set) of top-k documents d_r, along with $score(r, x)$, corresponding to a pattern with locus node x in GST. Clearly we cannot afford to maintain $\mathsf{top}(x, k)$ for all possible x's and k's. Rather, we will maintain the lists $\mathsf{top}(x, z)$ only for marked nodes x's (for various g values) and for k's that are powers of 2. Then $\mathsf{top}(x, k)$ for any x and k will be efficiently computed using that sampled data. The next section describes how we store and retrieve the sampled lists.

5.3 Storing and Retrieving the Lists $\mathsf{top}(x, z)$

The following is a key result in our scheme.

Lemma 22. *Let $g_h = z(\log^{(h)} n)^2$ for any $1 \leq h < \log^* n$, where $\log^{(1)} n = \log n$, $\log^{(h)} n = \log(\log^{(h-1)} n)$, and $\log^{(\log^* n)} n \leq 1$. Then $\mathsf{top}(x, z)$ for all $x \in S_{g_h}$ can be encoded in $s_h = s_{h-1} + O(n/\log^{(h)} n)$ bits, and $\mathsf{top}(x, z)$ for any given $x \in S_{g_h}$ can be decoded in time $t_h = t_{h-1} + O(z)$, where $s_1 = O(n/\log n)$ and $t_1 = O(z)$.*

Proof. We use induction. Consider the base case $h = 1$. For every $x \in S_{g_1}$, we maintain the list $\mathsf{top}(x, z)$ explicitly (using $O(\log n)$ bits per element), along with a pointer to the location where it is stored, in $s_1 = O(|S_{g_1}| z \log n) = O(n/\log n)$ bits. Thus the list $\mathsf{top}(x, z)$, for any $x \in S_{g_1}$, can be decoded in time $t_1 = O(z)$.

Now consider the grouping factor is g_h for $h \geq 2$. As we cannot afford to use $\Theta(\log n)$ bits per element, we introduce encoding schemes that reduce it to $O(\log^{(h)} n)$ bits. Thus the overall space for maintaining $\mathsf{top}(x, z)$ (in encoded form) for all $x \in S_{g_h}$ can be bounded by $O(|S_{g_h}| z \log^{(h)} n) = O(n/\log^{(h)} n)$ bits. Instead of using pointers as in the base case, we mark in a bitmap $B^h[1..2n]$ the node preorders of GST that belong to S_{g_h}. Therefore the list $\mathsf{top}(x, z)$ of a node $x \in S_{g_h}$ is stored in an array at offset $rank_{B^h}[x]$. Since we will only compute $rank$ on positions x where $B^h[x] = 1$, an "indexed dictionary" is sufficient [61], which requires $O((n/g_h) \log g_h + \lg \lg n) = o(n/\log^{(h)} n)$ bits and computes $rank$ in time $O(1)$. We now show how to encode the list $\mathsf{top}(x, z)$, for $x \in S_{g_h}$, in $O(\log^{(h)} n)$ bits per element, and how to decode it in $t_{h-1} + O(z)$ time.

We will maintain a structure STR_h, using s_h bits, for each grouping factor g_h, and will decode $\mathsf{top}(x, z)$ for $x \in S_{g_h}$ recursively, using $O(z)$ time in addition to the time needed to decode $\mathsf{top}(y, z)$ for some $y \in S_{g_{h-1}}$, as suggested in Lemma 22. As we cannot afford to sort the documents within the targeted query time, it is important to assume a fixed arrangement of documents within any particular decoded list $\mathsf{top}(\cdot, \cdot)$. That is, each time we decode a specific list, the decoding algorithm must return the elements in the same order.

Let x be a node in S_{g_h} and y (if it exists) be its highest descendant node in $S_{g_{h-1}}$. We show how to encode and decode $\mathsf{top}(x, z)$. To decode $\mathsf{top}(x, z)$, we first decode the list $\mathsf{top}(y, z)$ using STR_{h-1} in time t_{h-1}. From now onwards we have constant-time access to any element the list $\mathsf{top}(y, z)$. The the list $\mathsf{top}(x, z)$ will be partitioned into the following two disjoint lists:

(i) D_{old}, the documents that are common to $\mathsf{top}(x, z)$ and $\mathsf{top}(y, z)$.

(ii) D_{new}, the documents that are present in $\mathsf{top}(x, z)$, but not in $\mathsf{top}(y, z)$.

Encoding and Decoding Document Identifiers in D_{old}. We maintain a bit vector $B'[1..z]$, where $B'[i] = 1$ iff the ith document in $\mathsf{top}(y, z)$ is present in $\mathsf{top}(x, z)$. Therefore D_{old} can be decoded by listing those elements in $\mathsf{top}(y, z)$ (in the same order as they appear) at positions i where $B'[i] = 1$. Thus space for maintaining the encoded information is z bits and the time for decoding is $O(z)$.

Encoding and Decoding Document Identifiers in D_{new}. For each document $d_r \in D_{new}$, there exists at least one leaf in $Leaf(x \backslash y)$ that is marked with d_r (otherwise $score(r, x) = score(r, y)$ and d_r could not be in $\mathsf{top}(x, z)$ and not in $\mathsf{top}(y, z)$). Therefore, instead of explicitly storing r, it is sufficient to refer to such a leaf. For this we shall store a bit vector $B''[1..|L(x \backslash y)|]$ with all its bits in 0, except for $|D_{new}|$ 1's: for every document $d_r \in D_{new}$, we set one bit, say $B''[i] = 1$, where the ith leaf in $Leaf(x \backslash y)$ is

marked with d_r. Since $|B''| = |L(x\backslash y)| < 2g_{h-1}$ and the number of 1's is at most z, B'' can be encoded in $O(z \log(g_{h-1}/z)) = O(z \log^{(h)} n)$ bits with constant time $select$ support [61] ($select_{B''}(j)$ is the position of the j-th 1 in B''). Now, given B'', the documents in D_{new} can be identified in $O(z)$ time as follows: Find all those (at most z) increasing positions i where $B''[i] = 1$ using $select$ queries. Then, for each such i, find the ith leaf $\ell_{i'} \in L(x\backslash y)$ in constant time using the tree operations. Finally, report $d_{E\ [i']}$ as a document in D_{new} for each such i' using a constant-time $access$ operation on the document array.

As mentioned before, it is important for our (recursive) encoding/decoding algorithm to assume a fixed permutation of elements within any list $\mathsf{top}(\cdot, \cdot)$. We use the convention that, in $\mathsf{top}(x, z)$, the documents in D_{old} come before the documents in D_{new}. Moreover the documents within D_{old} and D_{new} are in the same order as the decoding algorithm identified them. In conclusion, the list of identifiers of documents in $\mathsf{top}(x, z)$ can be encoded in $O(z \log^{(h)} n)$ bits and decoded in $O(z)$ time, assuming constant-time access to any element in $\mathsf{top}(y, z)$. If node y does not exist, we proceed as if $\mathsf{top}(y, z) = \emptyset$ and $\mathsf{top}(x, z) = D_{new}$. We now consider how to encode the $score$'s associated with the elements in $\mathsf{top}(x, z)$ (i.e., $score(r, x)$ for all $d_r \in F(x, z)$).

Encoding and Decoding of Scores. Let d_{r_i}, for $i \in [1..z]$, be the ith document in $\mathsf{top}(x, z)$, and $f_i = score(r_i, x)$. Then, define $\delta_i = f_i - f'_i \geq 0$, where

$$f'_i = \begin{cases} score(r_i, y) & \text{if } i \leq |D_{old}| \text{ (i.e., if } r_i \in D_{old}), \\ \tau = \min\{score(r, y), r \in F(y, z)\} & \text{if } i > |D_{old}| \text{ (i.e., if } r_i \in D_{new}). \end{cases}$$

The following is an important observation: The number of leaves in $Leaf(x\backslash y)$ marked with document d_{r_i} is $score(r_i, x) - score(r_i, y)$, which is same as δ_i for $i \leq |D_{old}|$. For $i > |D_{old}|$, $score(r_i, x) - score(r_i, y) \geq \delta_i$, otherwise $score(r_i, y) > \tau$ and d_{r_i} would have qualified as a top-z document in $\mathsf{top}(y, z)$ (which is a contradiction as $d_{r_i} \in D_{new}$). By combining with the fact that each leaf node is marked with a unique document, we have the inequality $\sum_{i=1}^{z} \delta_i \leq |L(x\backslash y)| < 2g_{h-1}$. Therefore, δ_i for all $i \in [1..z]$ can be encoded using a bit vector $B''' = 10^{\delta_1}10^{\delta_2}10^{\delta_3} \ldots 10^{\delta_z}$ of length at most $2g_{h-1} + z$ with z 1's, in $O(z \log(g_{h-1}/z)) = O(z \log^{(h)} n)$ bits with constant-time $select$ support (refer to Section 2.7).

The decoding algorithm is described as follows: compute the f'_i's for $i = 1 \ldots z$ in the ascending order of i. For $i \leq |D_{old}|$, f'_i is given by score associated with the $(select_{B'}[i])$th document (which is same as d_{r_i}) in $\mathsf{top}(y, z)$. This takes only $O(z)$ time as the number of constant-time $select$ operations is $O(z)$, and we have

constant-time access to any element and score in $\mathsf{top}(y, z)$. Next, $\tau = \min\{score(r, y), r \in F(y, z)\}$ can be obtained by scanning the list $\mathsf{top}(y, z)$ once. Thus all the f'_i's are computed in $O(z)$ time. Next we decode each δ_i and add it to f'_i to obtain f_i, for $i = 1 \ldots z$ in $O(z)$ time, where $\delta_i = select_{B'''}(i) - select_{B'''}(i-1) - 1$ is computed in $O(1)$ time. Thus the space for maintaining the scores is $O(z \log^{(h)} n)$ bits and the time for decoding them is $O(z)$.

Adding over the h levels, the total space is $s_h = s_{h-1} + O(n/\log^{(h)} n) = O(n/\log^{(h)} n)$ bits and the total decoding time is $t_h = t_{h-1} + O(z) = O(zh)$ (note that $s_1 = O(n/\log n)$ and $t_1 = O(z)$). This completes the proof. □

5.4 Completing the Picture

Let $\pi \in [1..\log^* n)$ be an integer such that $\log^{(\pi-1)} n \geq \sqrt{\log^* n} > \log^{(\pi)} n$, then $\log^{(\pi)} n = \omega(1)$ (note that $\pi = \log^* n - \log^* \sqrt{\log^* n} = \Theta(\log^* n)$). Then, by choosing g_π as the grouping factor, the space s_π is $O(n/\log^{(\pi)} n) = o(n)$ bits. We maintain $\log D$ such structures corresponding to $z = 1, 2, 4, 8, ..., 2^{\lfloor \log D \rfloor}$, in $o(n \log D)$ bits total space. By combining the space bounds of all the components, we obtain the following lemma.

Lemma 23. *The total space requirement of our data structure is* $|\mathsf{CSA}| + n \log D + o(n \log D)$ *bits.*

The next lemma gives the total time to extract the sampled results and hints how we will use them.

Lemma 24. *Given any node* q *in* GST *and an integer* k*, our data structure can report the list* $\mathsf{top}(q', k)$ *in* $O(k \log^* n)$ *time, where* q' *is a node in the subtree of* q *with* $|L(q \backslash q')| = O(k\sqrt{\log^* n})$.

Proof. As the first step, round k to $z = 2^{\lceil \log k \rceil}$, which is the next highest power of 2. Then identify the highest node q', in the subtree of q, that is marked with respect to the grouping factor g_π: Let ℓ_i and ℓ_j be the leftmost and rightmost leaves of q in GST, then $q' = lca(\ell_{i'}, \ell_{j'})$ where $i' = g_\pi \cdot \lceil i/g_\pi \rceil$ and $j' = g_\pi \cdot \lfloor j/g_\pi \rfloor$ (there is no q' if $i' \geq j'$). This takes constant time on our representation of the GST topology.

Since $g_\pi = z \log^{(\pi)} n < z\sqrt{\log^* n}$, from Lemma 14 it holds $|L(q \backslash q')| = O(g_\pi) = O(z \log^{(\pi)} n) = O(k\sqrt{\log^* n})$. As $q' \in S_{g_\pi}$, the list $\mathsf{top}(q', z)$ can be decoded in time $t_\pi = O(z\pi) = O(z \log^* n)$ from the precomputed lists (from Lemma 22). The final $\mathsf{top}(q', k)$ can be obtained by filtering those documents in $\mathsf{top}(q', z)$ with score at least θ by a single scan of the list, where θ is the kth highest score in $\mathsf{top}(q', z)$ (which can be computed in $O(z) = O(k)$ time using the linear-time selection algorithm [7]). In case q' does not

exist, we report $\mathsf{top}(q', k) = \emptyset$, and even in such a case the inequality condition $|L(q)| < 2g_\pi$ is guaranteed (from Lemma 14). □

5.4.1 Query Answering

The query answering algorithm consists of the following steps:

- Find the locus node q of the input pattern P in GST by first obtaining the suffix range $[sp, ep]$ of P using CSA in $O(p)$ time, and then computing the lowest common ancestor of ℓ_{sp} and ℓ_{ep} in $O(1)$ time.
- Using Lemma 24, find the node q' in the subtree of q, where $|L(q \backslash q')| = O(k\sqrt{\log^* n})$ and retrieve the list $\mathsf{top}(q', k)$ in $O(k \log^* n)$ time.
- Every document d_r in the final output $\mathsf{top}(q, k)$ must either belong to $\mathsf{top}(q', k)$, or it must be that $r = E[i]$ for some leaf $\ell_i \in L(q \backslash q')$. Let us call S_{cand} the union of both sets of candidate documents. Then we compute $score(r, q)$ of each document $d_r \in S_{cand}$.
- Report k documents in S_{cand} with the highest $score(r, q)$ value. In this step, we first compute the kth highest score θ using the selection algorithm, and then use θ as a threshold for a document to be an output (more precisely, we report the $k' < k$ documents $d_r \in S_{cand}$ with $score(r, q) < \theta$ in a first pass, and then report the first $k - k'$ documents $d_r \in S_{cand}$ we find with $score(r, q) = \theta$ in a second pass). The time is $O(|S_{cand}|) = O(k\sqrt{\log^* n})$.

The overall time for Steps 1, 2, and 4 is $O(p + k \log^* n)$. In the remaining part of this section we show how to handle Step 3 efficiently as well, for the documents $r = E[i]$ we find in $Leaf(q \backslash q')$. Note that $score(r, q)$ can be computed as $rank_E(r, ep) - rank_E(r, sp - 1)$ using two $rank$ queries on the document array, but those $rank$ queries are expensive. Instead, we use a more sophisticated scheme where only the faster $select$, $access$, and partial $rank$ queries are used. This is described next.

5.4.2 Computing Scores Online

Firstly, we construct a supporting structure, SUP, in $O(k \log^* n)$ time and occupying $o(n \log D) + O(z \log n)$ bits, capable of answering the following query in $O(\log \log^* n)$ time: for any given r, return $score(r, q')$ if $r \in F(q', k)$, otherwise return -1. Let $\Delta = \Theta(\log^* n)$, then structure SUP is a forest of D/Δ balanced binary search trees $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_{D/\Delta}$. Initially each $\mathcal{T}_i$ is empty, hence the initial

space is $O(\log n)$ bits per tree (for maintaining a pointer to the location where it is stored), adding up to $O((D/\Delta)\log n) = o(n \log D)$ bits, which we consider a part of index. Next we shall insert each document $d_r \in F(q', k)$, along with its associated score, into tree $\mathcal{T}_{\lceil r/\Delta \rceil}$ of SUP. The size of each search tree can grow up to Δ, hence the total insertion time is $O(k \log \Delta)$. These insertions will increase the space of SUP by $O(k \log n)$ bits, which can be justified as it is the size of the output. Now we can search for any d_r in $\mathcal{T}_{r/\Delta}$ and, if $d_r \in F(q', k)$, we will retrieve $score(r, q')$ in $O(\log \Delta)$ time. Once we finish Step 3, these binary search trees can be set back to their initial empty state by visiting each document $d_r \in F(q', k)$ and deleting it from the corresponding tree in total $O(k \log \Delta)$ time. This does not impact the total asymptotic query processing time.

An outline of Step 3 follows: We scan each leaf $\ell_i \in L(q \backslash q')$, and compute $score(\cdot, q)$ of the corresponding document $d_{E\,[i]}$. Note that there can be many leaves in $Leaf(q \backslash q')$ marked with the same document, but we compute $score(\cdot, q)$ of a document only once (i.e., when we encounter it for the first time). After this, we also scan the documents $d_r \in F(q', k)$ and compute $score(r, q)$ if we have not considered this document in the previous step. However, the scanning of leaves is performed in a carefully chosen order. Let $\ell_{sp'}$ and $\ell_{ep'}$ be the leftmost and rightmost leaves in the subtree of q', and $B[1..D]$ be a bit vector initialized to all 0's (its size is D bits and can be considered a part of index). A detailed description of Step 3 follows:

- Start scanning the leaves ℓ_i for $i = sp, sp+1, \ldots, sp'-1$, in the *ascending* order of i, then for $i = ep, ep-1, \ldots, ep'+1$, in the *descending* order of i, and do the following: if $B[E\,[i]] = 0$, then set it to 1, compute $score(E\,[i], q)$, and store the result $(E\,[i], score(E\,[i], q))$ for Step 4. Note that each time we compute $score(E\,[i], q)$, i is either the first or the last occurrence of $E\,[i]$ in $E\,[sp, ep]$. Assume it is the first (the other case is symmetric). We use a constant-time partial $rank$ query, $x = rank_E\,(E\,[i], i)$. Then, by performing successive $select_E\,(E\,[i], j)$ queries for $j = x+1, x+2, \ldots, y$, where $select_E\,(E\,[i], y) > ep \geq select_E\,(E\,[i], y-1)$, we compute $score(E\,[i], q) = y - x$. The number of $select$ queries required is precisely $y - x = score(E\,[i], q)$, which can be further reduced as follows:

  - If $d_{E\,[i]} \in F(q', k)$, retrieve $score(E\,[i], q')$ from SUP in time $O(\log \log^* n)$. As we know that $score(E\,[i], q') \leq score(E\,[i], q)$, we start $select$ queries from $j = x + score(E\,[i], q')$, so

the number of $select$ queries used to find y is reduced to $score(E\ [i], q) - score(E\ [i], q') = score(E\ [i], L(q \backslash q'))$, that is, the number of leaves in $Leaf\,(q \backslash q')$ marked with $d_{E\ [i]}$.

– If $d_{E\ [i]} \notin F(q', k)$, compute $x' = select_E\ (E\ [i], x + \tau - 1)$, where we remind that $\tau = \min\{score(r, q'), r \in F(q', k)\}$. If $x' > ep$, we conclude that $score(E\ [i], q) < \tau$, and hence $d_{E\ [i]}$ can be discarded from being a candidate for the final output. On the other hand, if $x' \leq ep$, the $select$ queries can be started from $j = x + \tau$, which reduces the number of $select$ queries to $score(E\ [i], q) - \tau \leq score(E\ [i], L(q \backslash q'))$ (since $d_{E\ [i]} \notin F(q', k)$, it holds $score(E\ [i], q') \leq \tau$).

The query time for executing this step can be analyzed as follows: for each i, we perform a query on SUP. The computation of $score(E\ [i], q)$ requires at most $score(E\ [i], L(q \backslash q'))$ $select$ queries. As we do this computation only once per distinct document, the total number of $select$ queries is at most $\sum_r score(r, L(q \backslash q')) = |L(q \backslash q')|$. By choosing the cost $f(n, D) = \sqrt{\log^* n}$ for $select$ queries, the total time is $O(|L(q \backslash q')|(f(n, D) + \log\log^* n)) = O(k \log^* n)$.

- Now scan the documents $d_r \in F(q', k)$. If $B[r] = 0$, then there exists no leaf in $Leaf\,(q \backslash q')$ marked with d_r. Thus $score(r, q) = score(r, q')$ and the pair $(r, score(r, q'))$ is stored for Step 4. If $B[r] = 1$ then d_r has already been dealt with in the previous pass. The time for accessing $score(r, q')$ using SUP is $O(\log\log^* n)$, hence this step takes $O(k \log\log^* n)$ time.

- Reset B to its initial state (all bits set to 0) for supporting queries in future. By revisiting the leaves in $Leaf\,(q \backslash q')$ and the list $\mathsf{top}(q', k)$, we can exactly find out those locations in B where the corresponding bit is 1. The time for this step can be bounded by $O(|L(q \backslash q')| + k) = O(k\sqrt{\log^* n})$.

Thus the time for Step 3 is $O(k \log^* n)$, and the result follows.

5.5 Reducing the Time to $O(p + k \log^* k)$

Note that, when p or k is at least $\log\log n$, it already holds $O(p + k \log^* n) = O(p + k \log^* k)$. Therefore, we now concentrate on the case when $\max(p, k) < \log\log n$. We use the following result [28].

Lemma 25. *Given a fixed κ, an array $A[1..n]$ of n indices can be indexed in $O(n \log^2 \kappa)$ bits for answering the following query in $O(k)$ time, without accessing A and for any $1 \leq k \leq \kappa$: given i, j, and k, output the positions of the k highest elements in $A[i, j]$.*

Let S_δ be the set of nodes in GST with node depth equal to δ. We start with the description of an $O(n \log^2 \kappa)$-bit structure for a fixed $\kappa = \lg \lg n$ and a fixed $\delta < \lg \lg n$, for answering top-k queries for any $1 \leq k \leq \kappa$ and those patterns with their locus node belonging to S_δ. First, we construct an array $A[1..n]$ (with all its elements initialized to zero) as follows: For $i = 1 \ldots n$, if the first occurrence of document $E\,[i]$ in $E\,[a, b]$ is at position i, where $[a, b]$ is the suffix range corresponding to a unique node $u \in S_\delta$, then set $A[i] = score(E\,[i], u)$. We do not store this array explicitly, instead we maintain the structure of Lemma 25 over it, requiring $O(n \log^2 \kappa)$ bits space. Now the list of documents $\mathsf{top}(u, k)$ for any locus node $u \in S_\delta$ can be reported in $O(k)$ time as follows: First perform a top-k query on the structure of Lemma 25 with the suffix range $[sp, ep]$. The output will be a set of k locations $j_1, j_2, \ldots, j_k \in [sp, ep]$, and then the identifiers of the top-k documents are $E\,[j_1], E\,[j_2], \ldots, E\,[j_k]$. By maintaining similar structures for all the $\delta \in [1, \log \log n)$, any such top-$k$ query with $p < \lg \lg n$ can be answered in $O(p + k)$ time. The additional space required is $o(n(\log \log n)^3)$ bits, which can be bounded by $o(n \log \sigma)$ bits if, say, $\log \sigma \geq \sqrt{\log n}$. Otherwise, we shall explicitly maintain the top-κ documents corresponding to all patterns of length at most $\log \log n$, in decreasing frequency order, using a table of $O(\sigma^{\log \log n} \log \log n \log D) = o(n)$ bits. The query time in this case is just $O(k)$.

Thus, by combining the cases, we achieve $O(p + k \log^* k)$ query time.

Theorem 7. *There exists a compact index of* $|\mathsf{CSA}| + n \log D + o(n(\lg \sigma + \log D))$ *bits and near-optimal* $O(p + k \log^* k)$ *query time time, for the (unsorted) top-*k *frequent document retrieval problem.*

Concluding Remarks. We have shown that it is possible to obtain almost optimal time for top-k document retrieval, $O(p+k \log^* k)$, using compact space, $|\mathsf{CSA}|+n \lg D+o(n \lg D+n \log \sigma)$ bits. This is an important step towards answering the question of which is the minimum space that is necessary to obtain the optimal time, $O(p + k)$.

Chapter 6
Multipattern Retrieval

In this section, we consider a generalization of the top-k document retrieval problem. Instead of a single pattern P, a query now consists of a set $\mathcal{P} = \{P_1, P_2, \ldots, P_m\}$ of m patterns, and the relevance of a document d_r with respect to $\mathcal{P}$ depends only on the set of occurrences of all P_j in d_r. For simplicity, we first give an index for the simplest case, where $\mathcal{P}$ contains only two patterns P_1 and P_2 (of lengths p_1 and p_2, respectively).

We choose $\text{TF}(P_1, d_r)+\text{TF}(P_2, d_r)$ as the score function $score(P_1, P_2, d_r)$ with an additional restriction that in order for a document d_r to be qualified as an answer, both P_1 and P_2 must occur in d_r. Therefore, $score(P_1, P_2, d_r)$ is given by $\text{TF}(P_1, d_r)+\text{TF}(P_2, d_r)$ if both $\text{TF}(P_1, d_r)$,$\text{TF}(P_2, d_r) > 0$, and is zero otherwise. We later show how our index can be modified to handle other score functions.

Our index is built from the succinct framework in Section 4. It consists of a suffix array SA (of size $O(n)$ words) in addition to GST (uncompressed, whose size is $O(n)$ words), a document array E , and auxiliary structures for answering for top-z queries for fixed $z = 1, 2, 4, \ldots, D$. The auxiliary structure for a specific z can be constructed with $g = \sqrt{nz \log D}$ as the grouping factor, where we identify the marked nodes in GST. Note that the marked node information can be maintained in $O(n/g)$ bits (refer to Lemma 15). Let $\mathsf{top}(u, v, k)$ denote the list of top-z documents with respect to the score function $score(path(u), path(v), \cdot)$. Then, corresponding to all pairs of marked nodes u^* and v^* in GST, we maintain the list $\mathsf{top}(u^*, v^*, z)$ explicitly. The space for each specific auxiliary structure is thus bounded by $O(n/g) + O((n/g) \times (n/g) \times z \log D) = O(n)$ bits, so that the total space for all the $O(\log D)$ auxiliary structures is bounded by $O(n \log D) = O(n \log n)$ bits, which is $O(n)$ words.

Query Answering. The algorithm to answer a query is analogous to that of our succinct index in Section 4. First, we find the locus nodes u_{P_1} and u_{P_2} of P_1 and P_2, respectively, in $O(p_1 + p_2)$ time using GST. Next, we set $z = 2^{\lceil \log k \rceil}$ (the minimum power of 2 greater than or equal to the input integer k). Then, using the auxiliary structure specific to this z (with grouping factor $g = \sqrt{nz \log D}$), we find the highest marked descendent of nodes, $u^*_{P_1}$ and $u^*_{P_2}$, of the locus nodes u_{P_1} and u_{P_2}, respectively. Afterwards, the set

$$\mathsf{top}(u^*_{P_1}, u^*_{P_2}, z) \cup \{d_{E[i]} \mid \ell_i \in \mathit{Leaf}(u_{P_1} \backslash u^*_{P_1}) \cup \mathit{Leaf}(u_{P_2} \backslash u^*_{P_2})\}$$

will be a candidate set S_{cand} that contains the desired top k answers.

Hence, by computing $score(P_1, P_2, d_r)$ of each document $d_r \in S_{\mathsf{cand}}$, and by choosing those k highest-scoring documents, we obtain the final output. Given the suffix ranges of P_1 and P_2, the score of any particular document can be computed in $O(\log \log n)$ time using E (refer to Lemma 2 and Lemma 3, as $\text{TF}(P, d_r)$ can be evaluated by $rank_E$, given the suffix range of P, and $t_{sa} = O(1)$ when SA is stored explicitly). As $|S_{\mathsf{cand}}| = O(g + z)$, the overall query time can be bounded by $O(p_1 + p_2 + \sqrt{nk \log D} \log \log n)$.

Theorem 8. *A given collection $\mathcal{D}$ of D documents with n characters in total taken from an alphabet set $\Sigma = [\sigma]$ can be indexed in $O(n)$ words of space, such that whenever two patterns P_1 and P_2 (of p_1 and p_2 characters, respectively) and an integer k come as a query, the index returns those k documents with the highest $score(P_1, P_2, \cdot)$ values in decreasing order of $score(P_1, P_2, \cdot)$ in $O(p_1 + p_2 + \sqrt{nk \log D} \log \log n)$ time; here, $score(P_1, P_2, d_r) = \text{TF}(P_1, d_r) + \text{TF}(P_2, d_r)$ if both $\text{TF}(P_1, d_r)$ and $\text{TF}(P_2, d_r)$ are greater than 0, and is zero otherwise.*

The space of the index described in the above theorem can be easily made succinct by the following modifications: (i) replace GST by its compressed version (space required is $|\mathsf{CSA}| + O(n)$ bits), (ii) replace E by its compressed version as described in Lemma 2, and (iii) build the auxiliary structure by choosing a higher grouping factor of $g = \sqrt{nz} \log D$. The overall space occupancy can thus be bounded by $2|\mathsf{CSA}^*| + O(n)$ bits, however the query time will be increased to $O(t_s(p_1) + t_s(p_2) + \sqrt{nk} \log D \times t_{sa} \log \log n)$. We summarize the result in the following theorem.

Theorem 9. *A given collection $\mathcal{D}$ of D documents with n characters in total taken from an alphabet set $\Sigma = [\sigma]$ can be indexed in $2|\mathsf{CSA}^*| + O(n)$ bits of space, such that whenever two patterns P_1 and P_2 (of p_1 and p_2 characters, respectively) and an integer k come as a query, the index returns those k documents with the highest $score(P_1, P_2, \cdot)$ values in decreasing order of $score(P_1, P_2, \cdot)$ in $O(t_s(p_1) + t_s(p_2) + \sqrt{nk} \log D \times t_{sa} \log \log n)$ time; here, $score(P_1, P_2, d_r) = \text{TF}(P_1, d_r) + \text{TF}(P_2, d_r)$ if both $\text{TF}(P_1, d_r)$ and $\text{TF}(P_2, d_r)$ are greater than 0, and is zero otherwise, $|\mathsf{CSA}^*|$ denotes the maximum space (in bits) to store either a compressed suffix array (CSA) of the concatenated text with all the given documents in $\mathcal{D}$, or all the CSAs of individual documents, t_{sa} is the time decoding a suffix array value, $t_s(p)$ is the time for computing the suffix range of P using CSA.*

The above indexes can readily be adapted to handle the case with other score functions, with tradeoffs between the space for storing a data structure that can compute $score(\cdot, \cdot, \cdot)$ on the fly, and the per-document reporting time. In particular, the space remains $O(n)$ words for *linearly-calculable* score functions, where $score(\cdot, \cdot, d_r)$ can be computed on the fly by maintaining an $O(|d_r|)$-word index. For instance, when $docrank$ is the score function [1], the following result can be obtained.

Theorem 10. *A given collection $\mathcal{D}$ of D documents with n characters in total taken from an alphabet set $\Sigma = [\sigma]$ can be indexed in $O(n)$ words of space or in $2|\mathsf{CSA}^*| + O(n)$ bits of space, such that whenever two patterns P_1 and P_2 (of p_1 and p_2 characters, respectively) and an integer k come as a query, then among all those documents containing both P_1 and P_2, the index returns k documents with the highest $docrank(\cdot)$ values in decreasing order of $docrank(\cdot)$ in $O(p_1 + p_2 + \sqrt{nk \log D} \log\log D)$ time or in $O(t_s(p_1) + t_s(p_2) + \sqrt{nk} \log D \times t_{sa} \log\log n)$ time respectively; here, $docrank(d_r)$ of a document d_r is a static importance score associated with d_r, $|\mathsf{CSA}^*|$ denotes the maximum space (in bits) to store either a compressed suffix array (CSA) of the concatenated text with all the given documents in $\mathcal{D}$, or all the CSAs of individual documents, t_{sa} is the time decoding a suffix array value, $t_s(p)$ is the time for computing the suffix range of P using CSA.*

Remark. The index of Theorem 10 can be used to solve the document listing problem for two patterns, where the task is to report all those documents containing both the input patterns P_1 and P_2. To do so, we simply set $k = D$ and then obtain the output of each query in $O(p_1 + p_2 + \sqrt{nD \log D} \log\log D)$ time (assuming our linear space structure). To reduce the last term in the query bound, we can issue the top-1 query, then top-2, then top-4, and so on until a top-q query returns the $\mathtt{ndoc}$ answers, where $\mathtt{ndoc} < q$ denotes the number of documents in the desired output. Note that the patterns are searched only once here. Hence, the query time will be $O(p_1 + p_2 + \sqrt{n \log D} \log\log D + \sqrt{2n \log D} \log\log D + \sqrt{4n \log D} \log\log D + \cdots + \sqrt{n \times \mathtt{ndoc} \log D} \log\log D) = O(p_1 + p_2 + \sqrt{(\mathtt{ndoc} + 1) \times n \log D} \log\log D)$. Using similar analysis the time for query on the succinct space structure can be bounded by $O(t_s(p_1) + t_s(p_2) + \sqrt{n(\mathtt{ndoc} + 1)} \log D \times t_{sa} \log\log n)$. Notice that our result clearly improves the earlier solution for this problem by Cohen and Porat [12], where space occupancy and query time were $O(n \log n)$ words and $= O(p_1 + p_2 + \sqrt{(\mathtt{ndoc} + 1) \times n} \log^{2.5} D)$ respectively.

[1] For a document to be qualified as an output, both P_1 and P_2 must be present in it.

Theorem 11. *A given collection $\mathcal{D}$ of D documents with n characters in total taken from an alphabet set $\Sigma = [\sigma]$ can be indexed in $O(n)$ words of space or in $2|\mathsf{CSA}^*| + O(n)$ bits of space, such that whenever two patterns P_1 and P_2 (of p_1 and p_2 characters respectively) come as a query, the index returns all those* ndoc *documents containing both P_1 and P_2 in $O(p_1 + p_2 + \sqrt{(\texttt{ndoc}+1) \times n \log D} \log\log D)$ time or in $O(t_s(p_1) + t_s(p_2) + \sqrt{n(\texttt{ndoc}+1)} \log D \times t_{sa} \log\log n)$ time, here, $|\mathsf{CSA}^*|$ denotes the maximum space (in bits) to store either a compressed suffix array (*CSA*) of the concatenated text with all the given documents in $\mathcal{D}$, or all the* CSA*s of individual documents, t_{sa} is the time decoding a suffix array value, $t_s(p)$ is the time for computing the suffix range of P using* CSA*.* □

In another problem introduced by Muthukrishnan [46], the query asks for reporting all those documents with term proximity score at most an integer K, where P_1, P_2 and K and input parameters to the query, and term proximity score $\texttt{TP}_{two}(P_1, P_2, d_r)$ is defined as the distance between the closest occurrences of P_1 and P_2 within document d_r. If either P_1 or P_2, or both is absent in d_r, $\texttt{TP}_{two}(P_1, P_2, d_r)$ is infinity. An $O(n^{3/2} \log n)$-word index with $O(p_1 + p_2 + \sqrt{n} \log n + \texttt{output})$ query time was also proposed in [46], where output is the number of reported documents. Our framework can be used to derive the following linear space solution for the top-k version of this problem.

Theorem 12. *A given collection $\mathcal{D}$ of D documents with n characters in total taken from an alphabet set $\Sigma = [\sigma]$ can be indexed in $O(n)$ words, such that whenever two patterns P_1 and P_2 (of p_1 and p_2 characters, respectively) and an integer k come as a query, then among all those documents containing both P_1 and P_2, the index returns k documents with the* lowest $\texttt{TP}_{two}(P_1, P_2, \cdot)$ *values in increasing order of $\texttt{TP}_{two}(P_1, P_2, \cdot)$ in $O(p_1 + p_2 + \sqrt{nk \log D} \log^{\epsilon} n)$ time, where $\texttt{TP}_{two}(P_1, P_2, d_r)$ of a document d_r is the distance between the closest occurrences of P_1 and P_2 in d_r.*

Proof. The index construction is exactly same as that of the result in Theorem 8 except that we use a different score function here. Additionally we maintain an orthogonal range successor/predecessor search structure over the suffix array SA. For this, we use the $O(n)$-word space structure by Navarro and Nekrich [51]. Therefore, given any suffix range $[L, R]$ and a position pos as input, the smallest (resp., the largest) $SA[i]$ value, where $i \in [L, R]$, succeeding (resp., preceding) pos can be computed in $O(\log^{\epsilon} n)$ time, where $\epsilon > 0$ is any small constant. Using similar analysis as that of Theorem 8, the total space occupancy can be bounded by $O(n)$ words.

We use the same terminologies as that of the proof of Theorem 8. The only difference in query algorithm, compared to that of Theorem 8 is the way we compute $score$ of documents corresponding to $\{d_{E[i]} \mid \ell_i \in Leaf(u_{P_1} \backslash u^*_{P_1}) \cup Leaf(u_{P_2} \backslash u^*_{P_2})\}$. Let $[sp^*_1, ep^*_1]$ and $[sp^*_2, ep^*_2]$ represents the range of leaves in the subtree of $u^*_{P_1}$ and $u^*_{P_2}$ respectively. Assume that a document $d_{E[i]}$ is a top-k candidate, and that its candidacy is due to an occurrence of P_1 (resp., P_2) at position $SA[i]$, then the corresponding $\texttt{TP}_{two}(\cdot, \cdot, \cdot)$ value can be computed in $O(\log^\epsilon n)$ time by retrieving the successor and predecessor of $SA[i]$ with $[ep^*_2, ep^*_2]$ (resp., $[sp^*_1, ep^*_1]$) as the input range, so that we can find the distance from the closest occurrence of P_2 (resp., P_1) from it. Therefore, using similar analysis the time can be bounded by $O(p_1 + p_2 + \sqrt{nk \log D} \log^\epsilon n)$. □

6.1 Handling $m > 2$ Patterns

All the above results can be extended to handle the case where the query consists of a set of $m > 2$ patterns $\mathcal{P} = \{P_1, P_2, \ldots, P_m\}$, with p_i denoting the length of P_i. Precisely, for a specific 2-power z, we choose a grouping factor $g = n^{1-1/m}(z \log D)^{1/m}$, identify the marked nodes in GST, and maintain top-z documents corresponding to each combination of $(u^*_1, u^*_2, \ldots, u^*_m)$, where u^*_i for any i denotes a marked node in GST. Over all $\log D$ choices of z, the total space can be bounded by $O((n/g)^m z \log D) \times \log D = O(n \log n)$ bits, or equivalently by $O(n)$ words. Note that m is fixed at index construction time. Then, whenever a query comes, we can quickly find a candidate set S_{cand} of $O(n^{1-1/m}(k \log D)^{1/m})$ documents, compute the score of a document (if needed) in S_{cand} in $O(m \log \log D)$ time, and finally output the k highest-scoring ones among them. Putting everything together, we can obtain an $O(n)$-word index with query time $O(\sum_{i=1}^m p_i + mn^{1-1/m}(k \log D)^{1/m} \log \log D)$.

Chapter 7
Conclusions

We presented space-efficient frameworks for designing data structures for top-k string retrieval problems. Our frameworks are based on annotating suffix tree (or compressed suffix tree) with additional information. In particular, we maintain a suffix tree of the concatenated documents, superimpose the local suffix trees of the individual documents in terms of "links", and solve geometric range problems on these links. Our compressed framework samples these links as they pass through some specially chosen nodes. These frameworks are fairly general and have also been shown to be practical [58, 15, 49, 6]. Even though efficient solutions are already available for the central problem, there are still many interesting variations and open questions one could ask about. We conclude with some of them as listed below:

- Our I/O-optimal data structure requires $O(n \log^* n)$-word. It is interesting to see if we can bring down this space to linear (i.e., using $O(n)$ words) without sacrificing the optimality in the I/O bound. (See [59] for some recent results in this line of research) Designing data structure in the cache-oblivious model [22] is another future research direction.

- The current space-optimal data structure for top-k frequent document retrieval is proposed by Navarro and Thankachan [53]), whose per-document reporting time is $O(t_{sa} \log^2 k \log^\epsilon n)$. In contrast, the per-document reporting time of our compressed data structure (Theorem 4) is faster by a factor of $\log k$, but our data structure takes twice the size of text. An interesting problem is to design a space-optimal data structure, while keeping the query time the same as (or better than) that of ours (which is currently the fastest in compressed space).

- The *document selection* problem — where we want to obtain the kth highest-scoring document corresponding to the query — may have useful IR applications in practice (See [45] for some recent results on this problem).

- Another space-time tradeoff for parametrized top-k query [52]. For example, design an optimal query time data structure using $O(n \log^\epsilon n)$ words of space.

- A generalization of the multi-pattern problem may consist of forbidden patterns in the query. For instance, for the two-pattern case, one may want a query with patterns P_1 and P_2, and an integer k, where the task is to report the top k documents based on $score(P_1, \cdot)$ among all those documents d which do not contain the forbidden pattern P_2. Some progress has been made for the two-pattern case, but designing an efficient data structure to handle a set of forbidden patterns remains a challenge.

- Currently the gap between the upper and lower bound for two-pattern query problem is huge. It is interesting to see if this gap can be reduced. Can we obtain similar (or better) lower bounds for the forbidden pattern query problem. We strongly believe that the lower bounds for these problems are different from the currently known upper bounds [19, 31, 32] by $(\log n)^{O(1)}$ factor only (See [36] for some recent developments).

- Even though many succinct data structures have been proposed for top-k queries for frequency or PageRank-based score functions, it is still unknown if a succinct data structure with $O((p+k)\log^{O(1)} n)$ query time can be designed if the score function is term proximity (as it is not known to be succinctly calculable). Designing such a data structure even for special cases (say, with long query patterns only, or when we allow approximate score, etc.) or deriving lower bounds are interesting research directions. We remark that it is possible to design such a data structure for the special case where the input pattern is of length at least $\log^2 n$, by combining our succinct framework with known techniques [32, 10].

- Approximate pattern matching (i.e., allowing bounded errors and don't cares) is another active research area [13, 40]. Adding this aspect to document retrieval leads to many new problems. The following is one such problem: Report all those documents in which the edit (or Hamming) distance between one of its substrings and P is at most τ, where $\tau \geq 1$ is an input parameter.

- Building a data structure for a highly repetitive or a highly similar document collection is an active line of research. In recent work, Gagie et al. [24] propose an efficient document retrieval data structure suitable for a repetitive collection. An open problem is to extend the result for handling top-k queries.

Bibliography

[1] Peyman Afshani. On dominance reporting in 3d. In *Proceedings of European Symposium on Algorithms*, pages 41–51, 2008.

[2] Alok Aggarwal and Jeffrey Scott Vitter. The Input/Output Complexity of Sorting and Related Problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[3] Lars Arge, Vasilis Samoladas, and Jeffrey Scott Vitter. On two-dimensional indexability and optimal range search indexing. In *Proceedings of Symposium on Principles of Database Systems*, pages 346–357, 1999.

[4] D. Belazzougui and G. Navarro. Improved Compressed Indexes for Full-Text Document Retrieval. In *Proceedings of International Symposium on String Processing and Information Retrieval*, pages 386–397, 2011.

[5] Djamal Belazzougui and Gonzalo Navarro. New lower and upper bounds for representing sequences. In *Proceedings of European Symposium on Algorithms*, pages 181–192, 2012.

[6] Djamal Belazzougui, Gonzalo Navarro, and Daniel Valenzuela. Improved Compressed Indexes for Full-Text Document Retrieval. *Journal of Discrete Algorithms*, 18:3–13, 2013.

[7] Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Time Bounds for Selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973.

[8] G. S. Brodal, M. Greve R. Fagerberg, and A. López-Ortiz. Online sorted range reporting. In *Proceedings of International Symposium on Algorithms and Computation*, pages 173–182, 2009.

[9] Bernard Chazelle. Lower bounds for orthogonal range searching: I. the reporting case. *Journal of the ACM*, 37(2):200–212, 1990.

[10] Yu Feng Chien, Wing Kai Hon, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Geometric BWT: Compressed Text Indexing via Sparse Suffixes and Range Searching. *Algorithmica*, 2013. To appear.

[11] D. R. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, 1996.

[12] H. Cohen and E. Porat. Fast Set Intersection and Two-Patterns Matching. *Theoretical Computer Science*, 411(40–42):3795–3800, 2010.

[13] Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary Matching and Indexing with Errors and Don't Cares. In *Proceedings of Symposium on Theory of Computing*, pages 91–100, 2004.

[14] J. S. Culpepper, G. Navarro, S. J. Puglisi, and A. Turpin. Top-k Ranked Document Search in General Text Databases. In *Proceedings of European Symposium on Algorithms*, pages 194–205, 2010.

[15] J. Shane Culpepper, Matthias Petri, and Falk Scholer. Efficient in-memory top-k document retrieval. In *Proceedings of SIGIR Conference on Research and Development in Information Retrieval*, pages 225–234, 2012.

[16] P. Ferragina and R. Grossi. The String B-tree: A New Data Structure for String Searching in External Memory and Its Application. *Journal of the ACM*, 46(2):236–280, 1999.

[17] P. Ferragina and G. Manzini. Indexing Compressed Text. *Journal of the ACM*, 52(4):552–581, 2005.

[18] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed Representations of Sequences and Full-Text Indexes. *ACM Transactions on Algorithms*, 3(2), 2007.

[19] Johannes Fischer, Travis Gagie, Tsvi Kopelowitz, Moshe Lewenstein, Veli Mäkinen, Leena Salmela, and Niko Välimäki. Forbidden Patterns. In *Proceedings of Latin American Symposium on Theoretical Informatics*, pages 327–337, 2012.

[20] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a Sparse Table with $O(1)$ Worst Case Access Time. *Journal of the ACM*, 31(3):538–544, 1984.

[21] Michael L. Fredman and Dan E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci.*, 48(3):533–551, 1994.

[22] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-Oblivious Algorithms. In *Proceedings of Symposium on Foundations of Computer Science*, pages 285–298, 1999.

[23] T. Gagie, G. Navarro, and S. J. Puglisi. Colored Range Queries and Document Retrieval. In *Proceedings of International Symposium on String Processing and Information Retrieval*, pages 67–81, 2010.

[24] Travis Gagie, Kalle Karhu, Gonzalo Navarro, Simon J. Puglisi, and Jouni Sirén. Document Listing on Repetitive Collections. In *Proceedings of Symposium on Combinatorial Pattern Matching*, pages 107–119, 2013.

[25] Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Rank/Select Operations on Large Alphabets: A Tool for Text Indexing. In *Proceedings of Symposium on Discrete Algorithms*, pages 368–373, 2006.

[26] R. Grossi, A. Gupta, and J. S. Vitter. High-Order Entropy-Compressed Text Indexes. In *Proceedings of Symposium on Discrete Algorithms*, pages 841–850, 2003.

[27] R. Grossi and J. S. Vitter. Compressed Suffix Arrays and Suffix Trees with Applications to Text Indexing and String Matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

[28] Roberto Grossi, John Iacono, Gonzalo Navarro, Rajeev Raman, and Srinivasa Rao Satti. Encodings for range selection and top-k queries. In *Proceedings of European Symposium on Algorithms*, pages 553–564, 2013.

[29] Wing-Kai Hon, Manish Patil, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Indexes for document retrieval with relevance. In *Space-Efficient Data Structures, Streams, and Algorithms*, pages 351–362, 2013.

[30] Wing Kai Hon, Rahul Shah, and Sharma V. Thankachan. Towards an Optimal Space-and-Query-Time Index for Top-k Document Retrieval. In *Proceedings of Symposium on Combinatorial Pattern Matching*, pages 173–184, 2012.

[31] Wing-Kai Hon, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. String retrieval for multi-pattern queries. In *Proceedings of International Symposium on String Processing and Information Retrieval*, pages 55–66, 2010.

[32] Wing Kai Hon, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. On Position Restricted Substring Searching in Succinct Space. *Journal of Discrete Algorithms*, 17:109–114, 2012.

[33] Wing Kai Hon, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter. Space-efficient framework for top-k string retrieval. *Journal of the ACM*, 2014. To appear.

[34] Wing Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. Space-Efficient Framework for Top-k String Retrieval Problems. In *Proceedings of Symposium on Foundations of Computer Science*, pages 713–722, 2009.

[35] Wing Kai Hon, Sharma V. Thankachan, Rahul Shah, and Jeffrey Scott Vitter. Faster Compressed Top-k Document Retrieval. In *Proceedings of Data Compression Conference*, pages 341–350, 2013.

[36] Jesper Sindahl Nielsen Sharma V. Thankachan Kasper Green Larsen, J. Ian Munro. On hardness of several string problems. In *CPM*, 2014.

[37] D. E. Knuth, J. H. Morris, and V. B. Pratt. Fast Pattern Matching in Strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[38] Roberto Konow and Gonzalo Navarro. Faster Compact Top-k Document Retrieval. In *Proceedings of Data Compression Conference*, pages 351–360, 2013.

[39] Gregory Kucherov, Yakov Nekrich, and Tatiana A. Starikovskaya. Cross-document pattern matching. In *Proceedings of Symposium on Combinatorial Pattern Matching*, pages 196–207, 2012.

[40] Moshe Lewenstein, J. Ian Munro, Venkatesh Raman, and Sharma V. Thankachan. Less space: Indexing for queries with wildcards. In *Proceedings of International Symposium on Algorithms and Computation*, pages 89–99, 2013.

[41] U. Manber and G. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[42] Y. Matias, S. Muthukrishnan, S. C. Sahinalp, and J. Ziv. Augmenting Suffix Trees, with Applications. In *Proceedings of European Symposium on Algorithms*, pages 67–78, 1998.

[43] E. M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[44] J. I. Munro, V. Raman, and S. S. Rao. Space Efficient Suffix Trees. *Journal of Algorithms*, 39(2):205–222, 2001.

[45] J. Ian Munro, Gonzalo Navarro Rahul Shah, and Sharma V. Thankachan. Ranked document selection. 2014.

[46] S. Muthukrishnan. Efficient Algorithms for Document Retrieval Problems. In *Proceedings of Symposium on Discrete Algorithms*, pages 657–666, 2002.

[47] G. Navarro and V. Mäkinen. Compressed Full-Text Indexes. *ACM Computing Surveys*, 39(1), 2007.

[48] G. Navarro and Y. Nekrich. Top-k Document Retrieval in Optimal Time and Linear Space. In *Proceedings of Symposium on Discrete Algorithms*, pages 1066–1077, 2012.

[49] G. Navarro, S. J. Puglisi, and D. Valenzuela. Practical Compressed Document Retrieval. In *Proceedings of Symposium on Experimental Algorithms*, pages 193–205, 2011.

[50] Gonzalo Navarro. Spaces, Trees and Colors: The Algorithmic Landscape of Document Retrieval on Sequences. *CoRR*, abs/1304.6023, 2013.

[51] Gonzalo Navarro and Yakov Nekrich. Sorted range reporting. In *Proceedings of Symposium on Switching and Automata Theory*, pages 271–282, 2012.

[52] Gonzalo Navarro and Yakov Nekrich. Optimal Top-k Document Retrieval. *CoRR*, abs/1307.6789, 2013.

[53] Gonzalo Navarro and Sharma V. Thankachan. Faster top-k document retrieval in optimal space. In *Proceedings of International Symposium on String Processing and Information Retrieval*, pages 255–262, 2013.

[54] Gonzalo Navarro and Sharma V. Thankachan. Top-k document retrieval in compact space and near-optimal time. In *Proceedings of International Symposium on Algorithms and Computation*, pages 394–404, 2013.

[55] Gonzalo Navarro and Sharma V. Thankachan. New space/time tradeoffs for top-k document retrieval on sequences. 2014.

[56] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[57] Rasmus Pagh. Low Redundancy in Static Dictionaries with Constant Query Time. *SIAM Journal on Computing*, 31(2):353–363, 2001.

[58] M. Patil, S. V. Thankachan, R. Shah, W. K. Hon, J. S. Vitter, and S. Chandrasekaran. Inverted Indexes for Phrases and Strings. In *Proceedings of SIGIR Conference on Research and Development in Information Retrieval*, pages 555–564, 2011.

[59] Manish Patil, Sharma V. Thankachan, Rahul Shah, Yakov Nekrich, and Jeffrey Scott Vitter. Categorical range maxima queries. In *PODS*, 2014.

[60] Mihai Patrascu. Succincter. In *Proceedings of Symposium on Foundations of Computer Science*, pages 305–313, 2008.

[61] R. Raman, V. Raman, and S. S. Rao. Succinct Indexable Dictionaries with Applications to Encoding k-ary Trees, Prefix Sums and Multisets. *ACM Transactions on Algorithms*, 3(4), 2007.

[62] K. Sadakane. Succinct Data Structures for Flexible Text Retrieval Systems. *Journal of Discrete Algorithms*, 5(1):12–22, 2007.

[63] K. Sadakane and G. Navarro. Fully-Functional Succinct Trees. In *Proceedings of Symposium on Discrete Algorithms*, pages 134–149, 2010.

[64] Dekel Tsur. Top-k Document Retrieval in Optimal Space. *Information Processing Letters*, 113(12):440–443, 2013.

[65] N. Välimäki and V. Mäkinen. Space-Efficient Algorithms for Document Retrieval. In *Proceedings of Symposium on Combinatorial Pattern Matching*, pages 205–215, 2007.

[66] P. Weiner. Linear Pattern Matching Algorithms. In *Proceedings of Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[67] Dan E. Willard. Log-Logarithmic Worst-Case Range Queries are Possible in Space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 1983.

Vita

Sharma Valliyil Thankachan was born in Kerala, India, in 1984. He obtained his bachelor's degree in Electrical and Electronics and Engineering in 2006 from National Institute of technology (NIT), Calicut. During his doctoral studies at Louisiana State University, he has co-authored 32 refereed conference papers and 9 journal publications (published or accepted for publication by April 2014). His research interest falls in the area of algorithms and (space efficient) data structures. During the final year of his PhD, he received the Graduate School Dissertation Fellowship.